

A decorative border with intricate floral and scrollwork patterns in a light gray color, framing the central text.

Programa o em Python: Introdu

João Pavão Martins

VISIT...

LANZAROTE
Caliente.COM

A decorative border with intricate floral and scrollwork patterns, rendered in a light gray color, framing the central text.

Programa o em Python: Introdu

João Pavão Martins

Programação em Python:

Introdução à Programação

Utilizando Múltiplos Paradigmas

João Pavão Martins

Departamento de Engenharia Informática

Instituto Superior Técnico

Universidade Técnica de Lisboa

ii

Copyright ©2012 João

pavão

martins

Este livro não

pode ser reproduzido, no todo ou em parte, sob qualquer forma ou meio, electrónico ou mecânico, incluindo fotocópia, gravação,

ou qualquer sistema de registo de informação,

sem

autorização

prévia e escrita do autor:

João

pavão

martins

Departamento de Engenharia Informática

Instituto Superior Técnico

Av. Rovisco Pais

1096 Lisboa CODEX

Portugal

jpmartins@siscogl.pt

Índice

1 Computadores, algoritmos e programas

1	
1.1	Características de um computador
4	
1.2	Algoritmos
6	
1.2.1	Exemplos informais de algoritmos
6	
1.2.2	Características de um algoritmo
9	
1.3	Programas e algoritmos 11
1.3.1	Linguagens de programação 12
1.3.2	Exemplo de um programa 13
1.4	Sintaxe e semântica 16
1.4.1	Sintaxe 16
1.4.2	Semântica 24
1.4.3	Tipos de erros num programa 25
1.5	Notas finais 27
1.6	Exercícios 28

2 Elementos básicos de programação

31

2.1 Expressões 33

2.1.1 Constantes 33

2.1.2 Expressões compostas 35

2.2 Tipos elementares de informação 37

2.2.1 O tipo inteiro 39

2.2.2 O tipo real 39

2.2.3 O tipo lógico 43

2.3 Nomes e atribuição 44

2.4 Predicados e condições 51

2.5 Comunicação com o exterior 52

2.5.1 Leitura de dados 52

i

ii

ÍNDICE

2.5.2 Escrita de dados 55

2.6 Programas, instruções e sequenciação 57

2.7 Selecção 58

2.8 Repetição 63

2.9 Notas finais 67

2.10 Exercícios 67

3 Funções

71

3.1 Definição de funções em Python	73
3.2 Aplicação de funções em Python	75
3.3 Abstração procedimental	78
3.4 Exemplos	80
3.4.1 Nota final de uma disciplina	80
3.4.2 Potência	83
3.4.3 Factorial	83
3.4.4 Máximo divisor comum	84
3.4.5 Raiz quadrada	86
3.4.6 Seno	91
3.5 Estruturação de funções	94
3.6 Módulos	104
3.7 Notas finais	107
3.8 Exercícios	108

4 Tuplos e ciclos contados

111

4.1 Tuplos	111
4.2 Ciclos contados	119
4.3 Cadeias de caracteres revisitadas	123
4.4 Notas finais	133
4.5 Exercícios	133

5 Listas

135

5.1 Listas em Python	135
5.2 Métodos de passagem de parâmetros	139
5.2.1 Passagem por valor	140
5.2.2 Passagem por referência	142
5.3 O Crivo de Eratóstenes	143
5.4 Algoritmos de procura	146

ÍNDICE

iii

5.4.1 Procura sequencial	147
5.4.2 Procura binária	148
5.5 Algoritmos de ordenação	149
5.5.1 Ordenação por borbulhamento	152
5.5.2 Ordenação Shell	153
5.5.3 Ordenação por selecção	154
5.6 Exemplo	154
5.7 Considerações sobre eficiência	156
5.7.1 A notação do Omaiúsculo	160
5.8 Notas finais	162
5.9 Exercícios	163

6 Funções revisitadas

167

6.1 Funções recursivas	167
6.2 Funções de ordem superior	174
6.2.1 Funções como parâmetros	175
6.2.2 Funções como valor de funções	185
6.3 Programação funcional	190
6.4 Notas finais	193
6.5 Exercícios	193
7 Recursão e iteração	
197	
7.1 Recursão linear	198
7.2 Iteração linear	201
7.3 Recursão em processos e em funções	206
7.4 Recursão em árvore	207
7.4.1 Os números de Fibonacci	207
7.4.2 A torre de Hanói	212
7.5 Considerações sobre eficiência	216
7.6 Notas finais	218
7.7 Exercícios	218

8 Ficheiros

221

8.1 O tipo ficheiro	222
8.2 Leitura de ficheiros	223
8.3 Escrita em ficheiros	228
8.4 Notas finais	231

iv

ÍNDICE

8.5 Exercícios	231
--------------------------	-----

9 Dicionários

235

9.1 O tipo dicionário	235
9.2 Frequência de letras num texto	238
9.3 Dicionários de dicionários	242
9.4 Caminhos mais curtos em grafos	246
9.5 Notas finais	252
9.6 Exercícios	252

10 Abstracção de dados

255

10.1 A abstracção em programação	256
10.2 Motivação: números complexos	257
10.3 Tipos abstractos de informação	262
10.3.1 Identificação das operações básicas	263

10.3.2 Axiomatizaç~ao	266
10.3.3 Escolha da representaç~ao	266
10.3.4 Realizaç~ao das operaç~oes básicas	267
10.4 Barreiras de abstraç~ao	269
10.5 Objectos	272
10.6 Notas finais	279
10.7 Exercícios	279
11 Programaç~ao com objectos	
283	
11.1 O tipo conta bancária	284
11.2 Classes, subclasses e herança	287
11.3 Objectos em Python	296
11.4 Polimorfismo	297
11.5 Notas finais	301
11.6 Exercícios	302

12 O desenvolvimento de programas

303

12.1 A análise do problema	306
12.2 O desenvolvimento da solução	307
12.3 A programação da solução	309
12.3.1 A depuração	310
12.3.2 A finalização da documentação	313
12.4 A fase de testes	315

ÍNDICE

v

12.5 A manutenção	317
12.6 Notas finais	318

13 Estruturas lineares

321

13.1 Pilhas	321
13.1.1 Operações básicas para pilhas	322
13.1.2 Axiomatização	324
13.1.3 Representação de pilhas	325
13.2 Realização das operações básicas	326
13.2.1 Pilhas recorrendo a funções	326
13.2.2 A classe pilha	329
13.3 Expressões em notação pós-fixa	331
13.4 Balanceamento de parêntesis	335
13.5 Filas	337
13.5.1 Operações básicas para filas	337
13.5.2 Axiomatização	340
13.5.3 Representação de filas	340
13.5.4 A classe fila	341
13.6 Simulação de um supermercado	343
13.7 Representação gráfica	354
13.8 Notas finais	368
13.9 Exercícios	369
14 Árvores	

371

14.1 Operações básicas para árvores	373
---	-----

14.2 Axiomatizaç~ao	376
14.3 Representaç~ao de ~arvores	376
14.3.1 Representaç~ao para o uso de funç~oes	376
14.3.2 Representaç~ao para o uso com objectos	377
14.4 Realizaç~ao das operaç~oes b~asicas	378
14.4.1 ~Arvores recorrendo a funç~oes	379
14.4.2 A classe ~arvore	381
14.5 Ordenaç~ao por ~arvore	384
14.6 Notas finais	388
14.7 Exerc~icios	388

vi

ÍNDICE

15 Ep~ilogo

389

15.1 Programas	390
15.1.1 Algoritmos	391
15.1.2 Linguagens	393
15.1.3 Construç~ao de abstraç~oes	396
15.2 Programaç~ao	398
15.2.1 Arquitectura de programas	399
15.2.2 Paradigmas de programaç~ao	400
15.2.3 T~ecnicas usadas em programaç~ao	405
15.2.4 Sistemas operativos	409
15.3 Notas Finais	411

Prefácio

Este livro corresponde à matéria ensinada na disciplina semestral de Fundamentos da Programação da Licenciatura de Engenharia Informática e de Computadores do Instituto Superior Técnico. A disciplina situa-se no primeiro semestre do primeiro ano e não pressupõe conhecimentos prévios em programação. A matéria deste livro foi influenciada pelas diversas propostas internacionais relativas ao conteúdo de uma disciplina de introdução à programação a nível universitário [Koffmann et al., 1984], [Koffmann et al., 1985], [Turner, 1991],

[ACM, 2000], [ACM, 2008], [ACM, 2012] tendo sido também influenciada por um dos melhores livros existentes sobre programação, [Abelson et al., 1996].

O domínio da programação transcende o conhecimento de técnicas de programação, necessitando de uma compreensão profunda dos conceitos e da natureza da computação. Com este livro fornecemos uma abordagem disciplinada à actividade de programação e, simultaneamente, transmitimos conceitos genéricos relacionados com linguagens de programação. A escolha da linguagem de programação a utilizar numa disciplina introdutória é sempre um aspecto muito controverso. Na década de 1990, a linguagem utilizada pelo autor para ensinar introdução à programação foi o Pascal, na primeira década do século XXI, foi utilizada a linguagem Scheme. No ano lectivo de 2012/13 foi decidida a adopção da linguagem Python para leccionar a disciplina de Fundamentos da Programação. A nossa escolha foi influenciada por diversos factores:

— Apresentação de uma sintaxe mínima, facilitando a aprendizagem.

— Possibilidade de utilização de diversos paradigmas de programação, incluindo a programação imperativa, a programação funcional e a programação por objectos.

— Existência de processadores da linguagem, ao nível do domínio público, para vii

viii

ÍNDICE

os principais sistemas operativos, Macintosh, Windows e Unix.

Deve ser desde já clarificado que o objectivo deste livro não é o de ensinar Python (e consequentemente algumas das suas características não são abordadas), mas sim o de utilizar o Python para ensinar a programar em qualquer linguagem.

Existem vários aspectos que fazem com que este livro se distinga de outros livros que abordam a introdução à programação:

1. Apresenta a actividade de programação como uma construção de abstracções, tanto de abstracções de procedimentos, como de abstracções de dados.
2. Apresenta uma introdução geral à actividade de programação. O livro utiliza o Python como uma ferramenta para exprimir os conceitos introduzidos, sendo o seu objectivo a apresentação de conceitos essenciais de programação, tais como o desenvolvimento de algoritmos utilizando o método do topo para a base, a utilização de estruturas de informação adequadas, a abstracção procedimental e a abstracção de dados, estratégias para teste e depuração de programas, a documentação correcta e o anonimato da representação.
3. Enfatiza a distinção entre a sintaxe e a semântica. Para dominar a tarefa de programação não basta saber falar com a linguagem (escrever programas), é também necessário saber falar sobre a linguagem (utilizando português ou outra linguagem como metalinguagem). Este aspecto é abordado ao longo de todo o livro.
4. A metodologia para o desenvolvimento de tipos abstractos de informação é explicada em termos claros, e exemplos de tipos abstractos de informação são discutidos pormenorizadamente.

O Python é uma linguagem de programação inventada pelo cientista holandês Guido van Rossum no final da década de 1980. A sua primeira versão surgiu em 1989, tendo a versão utilizada neste livro, o Python 3, sido lançada em 2008.

Em 2007 e em 2010, o Python foi considerada a linguagem de programação do ano, sendo actualmente a 8ª linguagem mais popular, depois de C, Java, Objective-C, C++, C#, PHP e Visual BASIC. Entre as grandes organizações que utilizam o Python incluem-se a Google, a Yahoo!, o YouTube, o CERN e a NASA.

ÍNDICE

Agradecimentos

Gostava de agradecer a todas as pessoas que contribuíram para a forma final deste livro através de críticas e sugestões. Entre estas pessoas realço as seguintes (por ordem alfabética do último nome): Fausto Almeida, Vasco Almeida, Luís Manuel Tavares Jordão Alves, João Romeiras Amado, Ana Cardoso Cachopo, José Pedro Martins Cândido, Francisco Maria Calisto, João Carrapiço, Carlos Carvalho, João Gonçalo Ceia, Maria Teresa Coelho, João Colaço, Margarida Correia, Nuno Miguens Diogo, Adriana Domingos, Mariana Gaspar Fernandes, Alexandre Ferreira, Hugo Ferreira, Bernardo Pascoal Figueiredo, Francisco Castro Henriques, João Martins, Pedro Amaro de Matos, Francisco Melo, Diogo Filipe Panaca Mendes, Miguel Neves Pasadinhas, Pedro Pinela, Yuliya Plotka, Tiago Mendes de Almeida Rechau, João Filipe Coito Ribeiro, Vicente Rocha, Inês Santos, Márcio Santos, Anisa Shahidian, Daniel Sil, Daniel Tanque, João Vieira. Contudo, todos os erros e imperfeições que este livro indubitavelmente contém são da exclusiva responsabilidade do autor.

x

ÍNDICE

Capítulo 1

Computadores, algoritmos e

programas

“Take some more tea,” the March Hare said to Alice, very earnestly.

“I’ve had nothing yet,” Alice replied in an offended tone,

“so I can’t take more.”

“You mean you can’t take less,” said the Hatter: “it’s very easy to take more than nothing.”

Lewis Carroll, *Alice’s Adventures in Wonderland* Uma das características de um engenheiro é a capacidade para resolver problemas técnicos. A resolução deste tipo de problemas envolve uma combinação de ciência e de arte. Por ciência entende-se um conhecimento dos princípios matemáticos, físicos e dos aspectos técnicos que têm de ser bem compreendidos, para que sejam aplicados correctamente. Por arte entende-se a avaliação correcta, a experiência, o bom senso e o conhecimento que permitem representar

um problema do mundo real por um modelo ao qual o conhecimento técnico pode ser aplicado para produzir uma solução.

De um modo geral, qualquer problema de engenharia é resolvido recorrendo a uma sequência de fases: a compreensão do problema é a fase que corresponde a perceber e a identificar de um modo preciso o problema que tem de ser resolvido; após a compreensão do problema, entra-se na fase correspondente à especificação do problema, na qual o problema é claramente descrito e documentado, de modo 1

2

CAPÍTULO 1. COMPUTADORES, ALGORITMOS E PROGRAMAS

a remover dúvidas e imprecisões; no desenvolvimento da solução (ou modelação da solução) utiliza-se a especificação do problema para produzir um esboço da solução do problema, identificando métodos apropriados de resolução de problemas e as suposições necessárias; durante o desenvolvimento da solução, o esboço da solução é progressivamente pormenorizado até se atingir um nível de especificação que seja adequado para a sua realização; na concretização da solução, as especificações desenvolvidas são concretizadas (seja num objecto físico, por exemplo, uma ponte ou um processador, seja num objecto imaterial, por exemplo, um programa de computador); finalmente, na fase de verificações e testes, o resultado produzido é validado, verificado e testado.

A Engenharia Informática difere das engenharias tradicionais, no sentido em que trabalha com entidades imateriais. Ao passo que as engenharias tradicionais lidam com forças físicas, directamente mensuráveis (por exemplo, a gravidade, os campos eléctricos e magnéticos) e com objectos materiais que interagem com essas forças (por exemplo, rodas dentadas, vigas, circuitos), a Engenharia Informática lida com entidades intangíveis que apenas podem ser observadas indirectamente através dos efeitos que produzem.

A Engenharia Informática tem como finalidade a concepção e realização de abstrações ou modelos de entidades abstractas que, quando aplicadas por um computador, fazem com que este apresente um comportamento que corresponde à solução de um dado problema.

Sob a perspectiva da Informática que apresentamos neste livro, um computador é uma máquina cuja função é manipular informação. Por informação entende-se qualquer coisa que pode ser transmitida ou registada e que tem um significado associado à sua representação

simbólica. A informação pode ser transmitida de pessoa para pessoa, pode ser extraída directamente da natureza através de observação e de medida, pode ser adquirida através de livros, de filmes, da televisão, etc. Uma das características que distinguem o computador de outras máquinas que lidam com informação é o facto de este poder manipular a informação, para além de a armazenar e transmitir. A manipulação da informação feita por um computador segue uma sequência de instruções a que se chama um programa. Apesar de sabermos que um computador é uma máquina complexa, constituída por componentes electrónicos, nem os seus componentes nem a interligação entre eles serão aqui estudados. Para a finalidade que nos propomos atingir, o ensino da programação, podemos abstrair de toda a cons-

3

tituição física de um computador, considerando-o uma “caixa electrónica” que tem a capacidade de compreender e de executar programas.

A Informática é o ramo da ciência que se dedica ao estudo dos computadores e dos processos com eles relacionados: como se desenvolve um computador, como se especifica o trabalho a ser realizado por um computador, de que forma se pode tornar mais fácil de utilizar, como se definem as suas limitações e, principalmente, como aumentar as suas capacidades e o seu domínio de aplicação.

Um dos objectivos da Informática corresponde ao estudo e desenvolvimento de entidades abstractas geradas durante a execução de programas – os processos computacionais. Um processo computacional é um ente imaterial que evolui ao longo do tempo, executando acções que levam à solução de um problema.

Um processo computacional pode afectar objectos existentes no mundo real (por exemplo, guiar a aterragem de um avião, distribuir dinheiro em caixas multibanco, comprar e vender acções na bolsa), pode responder a perguntas (por exemplo, indicar quais as páginas da internet que fazem referência a um dado termo), entre muitos outros aspectos.

A evolução de um processo computacional é ditada por uma sequência de instruções a que se chama programa, e a actividade de desenvolver programas é chamada programação. A programação é uma actividade intelectual fascinante, que não é difícil, mas que requer muita disciplina. O principal objectivo deste livro é fornecer

uma introdução à programação disciplinada, ensinando os princípios e os conceitos subjacentes, os passos envolvidos no desenvolvimento de um programa e o modo de desenvolver programas bem estruturados, eficientes e sem erros.

A programação utiliza muitas actividades e técnicas que são comuns às utilizadas em projectos nos vários ramos da engenharia: a compreensão de um problema; a separação entre a informação essencial ao problema e a informação acessória; a criação de especificações pormenorizadas para o resolver; a realização destas especificações; a verificação e os testes.

Neste primeiro capítulo definimos as principais características de um computador e introduzimos um conceito essencial para a informática, o conceito de algoritmo.

4

CAPÍTULO 1. COMPUTADORES, ALGORITMOS E PROGRAMAS

1.1

Características de um computador

Um computador é uma máquina cuja função é manipular símbolos. Embora os computadores difiram em tamanho, aparência e custo, eles partilham quatro características fundamentais: são automáticos, universais, electrónicos e digitais.

Um computador diz-se automático no sentido em que, uma vez alimentado com a informação necessária, trabalha por si só, sem a intervenção humana. Não pretendemos, com isto, dizer que o computador começa a trabalhar por si só (necessita, para isso, da intervenção humana), mas que o computador procura por si só a solução dos problemas. Ou seja, o computador é automático no sentido em que, uma vez o trabalho começado, ele será levado até ao final sem a intervenção humana. Para isso, o computador recebe um programa, um conjunto de instruções quanto ao modo de resolver o problema. As instruções do programa são escritas numa notação compreendida pelo computador (uma linguagem de programação), e especificam exactamente como o trabalho deve ser executado. Enquanto o trabalho está a ser executado, o programa está armazenado dentro do computador e as suas instruções estão a ser seguidas.

Um computador diz-se universal, porque pode efectuar qualquer tarefa

cuja solução possa ser expressa através de um programa. Ao executar um dado programa, um computador pode ser considerado uma máquina orientada para um fim particular. Por exemplo, ao executar um programa para o tratamento de texto, um computador pode ser considerado como uma máquina para produzir cartas ou texto; ao executar um programa correspondente a um jogo, o computador pode ser considerado como uma máquina para jogar. A palavra

“universal” provém do facto de o computador poder executar qualquer programa, resolvendo problemas em diferentes áreas de aplicação. Ao resolver um problema, o computador manipula os símbolos que representam a informação pertinente para esse problema, sem lhes atribuir qualquer significado. Devemos, no entanto, salientar que um computador não pode resolver qualquer tipo de problema. A classe dos problemas que podem ser resolvidos através de um computador foi estudada por matemáticos antes da construção dos primeiros computadores. Durante a década de 1930, matemáticos como Alonzo Church (1903–1995), Kurt Gödel (1906–1978), Stephen C. Kleene (1909–1994), Emil Leon Post (1897–1954) e Alan Turing (1912–1954) tentaram definir matematicamente a classe das funções que podiam ser calculadas mecanicamente. Embora os métodos utilizados por estes matemáticos fossem muito diferentes, todos os

1.1. CARACTERÍSTICAS DE UM COMPUTADOR

5

formalismos desenvolvidos são equivalentes, no sentido em que todos definem a mesma classe de funções, as funções recursivas parciais. Pensa-se, hoje em dia, que as funções recursivas parciais são exactamente as funções que podem ser calculadas através de um computador. Este facto é expresso através da tese de Church-Turing¹.

De acordo com a tese de Church-Turing, qualquer computação pode ser baseada num pequeno número de operações elementares. Nos nossos programas, estas operações correspondem fundamentalmente às seguintes: 1. Operações de entrada de dados, as quais obtêm valores do exterior do programa;

2. Operações de saída de dados, as quais mostram valores existentes no programa;

3. Operações matemáticas, as quais efectuem cálculos sobre os dados existentes no programa;

4. Execução condicional, a qual corresponde ao teste de certas

condições e à execução de instruções, ou não, dependendo do resultado do teste; 5. Repetição, a qual corresponde à execução repetitiva de certas instruções.

A tarefa de programação corresponde a dividir um problema grande e complexo, em vários problemas, cada vez menores e menos complexos, até que esses problemas sejam suficientemente simples para poderem ser expressos em termos de operações elementares.

Um computador é electrónico. A palavra “electrónico” refere-se aos componentes da máquina, componentes esses que são responsáveis pela grande velocidade das operações efectuadas por um computador.

Um computador é também digital. Um computador efectua operações sobre informação que é codificada recorrendo a duas grandezas discretas (tipicamente referidas como sendo 0 e 1) e não sobre grandezas que variam de um modo contínuo. Por exemplo, num computador o símbolo “J”, poderá ser representado por 1001010.

Uma discussão sobre a tese de Church-Turing e sobre as funções recursivas parciais está para além da matéria deste livro. Este assunto pode ser consultado em [Brainerd and Landwebber, 1974], [Hennie, 1977] ou [Kleene, 1975].

6

CAPÍTULO 1. COMPUTADORES, ALGORITMOS E PROGRAMAS

1.2

Algoritmos

Ao apresentarmos as características de um computador, dissemos que durante o seu funcionamento ele segue um programa, um conjunto de instruções bem definidas que especificam exactamente o que tem que ser feito. Este conjunto de instruções é caracterizado matematicamente como um algoritmo². Os algoritmos foram estudados e utilizados muito antes do aparecimento dos computadores modernos. Um programa corresponde a um algoritmo escrito numa linguagem que é entendida pelo computador, chamada uma linguagem de programação.

Um algoritmo é uma sequência finita de instruções bem definidas e não ambíguas, cada uma das quais pode ser executada mecanicamente num período de tempo finito com uma quantidade de esforço finita.

Antes de continuar, vamos analisar a definição de algoritmo que acabámos de apresentar. Em primeiro lugar, um algoritmo consiste numa sequência finita de instruções. Isto quer dizer que existe uma ordem pela qual as instruções aparecem no algoritmo, e que estas instruções são em número finito. Em segundo lugar, as instruções de um algoritmo são bem definidas e não ambíguas, ou seja, o significado de cada uma das instruções é claro, não havendo lugar para múltiplas interpretações do significado de uma instrução. Em terceiro lugar, cada uma das instruções pode ser executada mecanicamente, isto quer dizer que a execução das instruções não requer imaginação por parte do executante.

Finalmente, as instruções devem ser executadas num período de tempo finito e com uma quantidade de esforço finita, o que significa que a execução de cada uma das instruções termina.

Um algoritmo está sempre associado com um dado objectivo, ou seja, com a solução de um dado problema. A execução das instruções do algoritmo garante que o seu objectivo é atingido.

1.2.1

Exemplos informais de algoritmos

A descrição de sequências de acções para atingir objectivos tem um papel fundamental na nossa vida quotidiana e está relacionada com a nossa facilidade 2A palavra “algoritmo” provém de uma variação fonética da pronúncia do último nome do matemático persa Abu Ja’far Mohammed ibn-Musa al-Khowarizmi (c. 780–c. 850), que desenvolveu um conjunto de regras para efectuar operaçõ

ões aritméticas com números decimais.

Al-Khowarizmi foi ainda o criador do termo “Álgebra” (ver [Boyer, 1974], páginas 166–167).

1.2. ALGORITMOS

7

de comunicar. Estamos constantemente a transmitir ou a seguir sequências de instruções, por exemplo, para preencher impressos, para operar máquinas, para nos deslocarmos para certo local, para montar objectos, *etc.*

Começamos por examinar algumas sequências de instruções

utilizadas na nossa vida quotidiana. Consideremos, em primeiro lugar, a receita de “Rebuçados de ovos”³:

REBUC

ADOS DE OVOS

500 g de açúcar;

2 colheres de sopa de amêndoas peladas e raladas; 5 gemas de ovos;

250 g de açúcar para a cobertura;

e farinha.

Leva-se o açúcar ao lume com um copo de água e deixa-se ferver até fazer ponto de pérola. Junta-se a amêndoa e deixa-se ferver um pouco. Retira-se do calor e adicionam-se as gemas. Leva-se o preparado novamente ao lume e deixa-se ferver até se ver o fundo do tacho. Deixa-se arrefecer completamente. Em seguida, com a ajuda de um pouco de farinha, molda-se a massa de ovos em bolas. Leva-se o restante açúcar ao lume com 1 dl de água e deixa-se ferver até fazer ponto de rebuçado. Passam-se as bolas de ovos por este açúcar e p

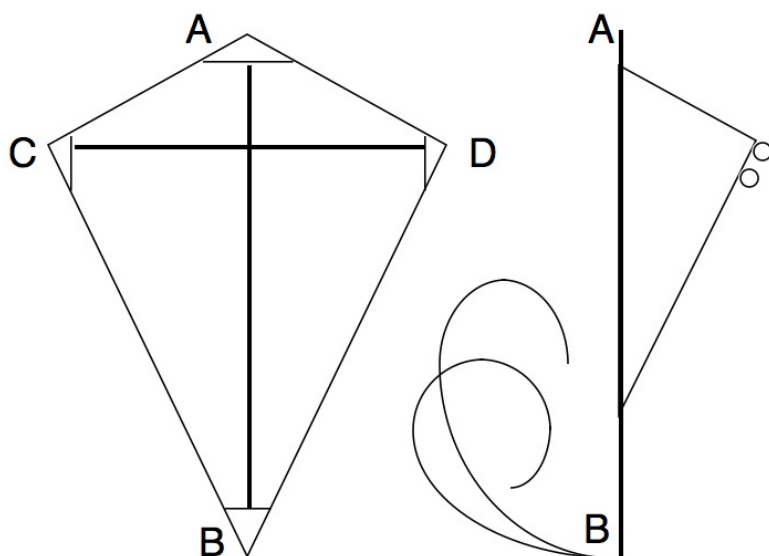
õem-se a secar sobre uma pedra untada, após o que se embrulham em papel celofane de várias cores.

Esta receita é constituída por duas partes distintas: (1) uma descrição dos objectos a manipular; (2) uma descrição das acções que devem ser executadas sobre esses objectos. A segunda parte da receita é uma sequência finita de instruções bem definidas (para uma pessoa que saiba de culinária e portanto entenda o significado de expressões como “ponto de pérola”, “ponto de rebuçado”, etc., todas as instruções desta receita são perfeitamente definidas), cada uma das quais pode ser executada mecanicamente (isto é, sem requerer imaginação por parte do executante), num período de tempo finito e com uma quantidade de esforço finita. Ou seja, a segunda parte desta receita é um exemplo informal de um algoritmo.

Consideremos as instruções para montar um papagaio voador, as quais estão associadas ao diagrama representado na Figura 1.14.

³De [Modesto, 1982], página 134. Reproduzida com autorização da Editorial Verbo.

⁴Adaptado das instruções para montar um papagaio voador oferecido



8

CAPÍTULO 1. COMPUTADORES, ALGORITMOS E PROGRAMAS

Figura 1.1: Diagrama para montar o papagaio voador.

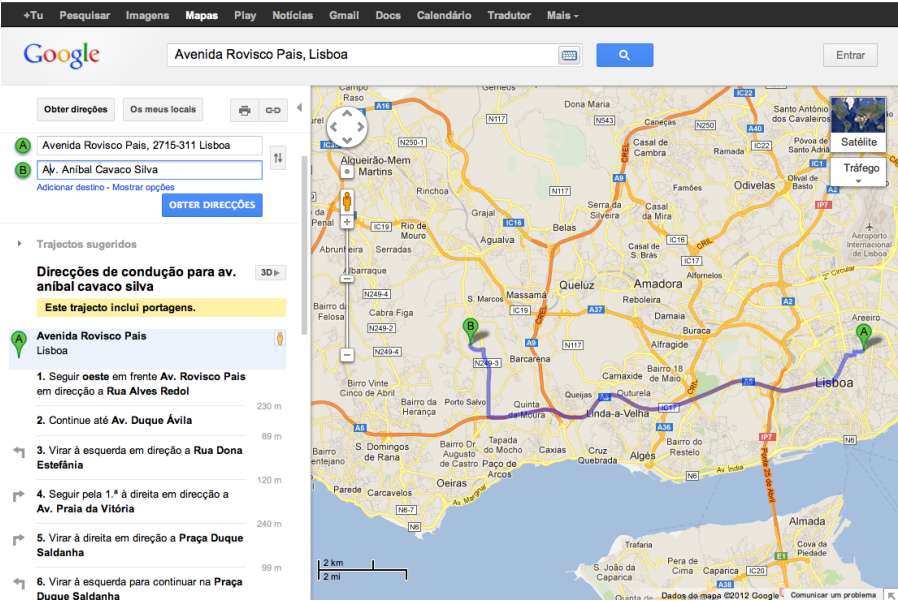
PAPAGAIO VOADOR

1. A haste de madeira central já se encontra colocada com as pontas medidas nas bolsas A. e B.
2. Dobre ligeiramente a haste transversal e introduza as suas extremidades nas bolsas C. e D.
3. Prenda as pontas das fitas da cauda à haste central no ponto B.
4. Prenda com um nó a ponta do fio numa das argolas da aba do papagaio. Se o vento soprar forte deverá prender na argola inferior. Se o vento soprar fraco deverá prender na argola superior.

As instruções para montar o papagaio voador são constituídas por uma descrição implícita dos objectos a manipular (mostrados na Figura 1.1) e por uma sequência de passos a seguir. Tal como anteriormente, estas instruções podem ser descritas como um

algoritmo informal.

Suponhamos que desejamos deslocar-nos do Instituto Superior Técnico na Ave-nida Rovisco Pais (campus da Alameda) para o campus do Tagus Parque (na Av. An íbal Cavaco Silva em Oeiras). Recorrendo ao Google Maps, obtemos a descriç ão apresentada na Figura 1.2. Nesta figura, para além de um mapa ilustrativo, aparecem no lado esquerdo uma sequência detalhada de instruções gal.



1.2. ALGORITMOS

9

Figura 1.2: Instruções para ir do Campus da Alameda ao do Tagus Parque.

do percurso a seguir para a deslocação pretendida. Novamente, estas instruções podem ser consideradas como um algoritmo informal.

1.2.2

Características de um algoritmo

Existem inúmeros exemplos de algoritmos que utilizamos na nossa vida quotidiana (instruções para fazer uma chamada telefónica, para nos deslocarmos para um certo local, etc.). Estes exemplos mostram que usamos algoritmos desde sempre: eles correspondem a sequências finitas de instruções que devem ser seguidas de modo a atingir um

determinado objectivo.

A sequência de passos de um algoritmo deve ser executada por um agente, o qual pode ser humano, mecânico, electrónico, ou qualquer outra coisa. Cada algoritmo está associado a um agente (ou a uma classe de agentes) que deve executar as suas instruções. Aquilo que representa um algoritmo para um agente pode não o ser para outro agente. Por exemplo, as instruções da receita dos rebuçados de ovos são um algoritmo para quem sabe de culinária e não o são para quem não sabe.

10

CAPÍTULO 1. COMPUTADORES, ALGORITMOS E PROGRAMAS

Embora um algoritmo não seja mais do que uma descrição da sequência de passos a seguir para atingir um determinado objectivo, nem todas as sequências de passos para atingir um dado objectivo podem ser consideradas um algoritmo, pois um algoritmo deve possuir três características, ser rigoroso, ser eficaz e ter a garantia de terminar.

1. Um algoritmo é rigoroso. Cada instrução do algoritmo deve especificar exacta e rigorosamente o que deve ser feito, não havendo lugar para ambiguidade. O facto de um algoritmo poder ser executado mecanicamente obriga a que cada uma das suas instruções tenha uma e só uma interpretação. Por exemplo, a instrução contida na receita dos rebuçados de ovos “leva-se o açúcar ao lume com um copo de água” pode ter várias interpretações. Uma pessoa completamente ignorante de processos culinários pode ser levada a colocar um copo de água (objecto de vidro) dentro de uma panela (ou sobre o lume, interpretando a frase à letra) juntamente com o açúcar.

Para evitar a ambiguidade inerente à linguagem utilizada pelos seres humanos (chamada linguagem natural, de que o português é um exemplo) criaram-se novas linguagens (chamadas linguagens artificiais) para exprimir os algoritmos de um modo rigoroso. Como exemplos de linguagens artificiais, já conhecemos a notação matemática, a qual permite escrever frases de um modo compacto e sem ambiguidade, por exemplo,

$\forall x \exists y : y > x$, e a notação química, que permite descrever compostos e reacções químicas de um modo compacto e não ambíguo, por exemplo, $\text{MgO} + \text{H}_2 \rightarrow \text{Mg} + \text{H}_2\text{O}$. A linguagem Python, discutida neste livro, é mais um exemplo de uma linguagem artificial.

2. Um algoritmo é eficaz. Cada instrução do algoritmo deve ser suficientemente básica e bem compreendida de modo a poder ser executada num intervalo de tempo finito, com uma quantidade de esforço finita. Para ilustrar este aspecto, suponhamos que estávamos a consultar as instruções que apareciam na embalagem do adubo “Crescimento Gigantesco”, as quais incluía a seguinte frase: “Se a temperatura máxima do mês de Abril for superior a 23°, misture o conteúdo de duas embalagens em 5 litros de água, caso contrário, misture apenas o conteúdo de uma embalagem”. Uma vez que não é difícil determinar qual a temperatura máxima do mês de Abril, podemos decidir se deveremos utilizar o conteúdo de duas embalagens ou

1.3. PROGRAMAS E ALGORITMOS

11

apenas o conteúdo de uma. Contudo, se o texto fosse: “Se a temperatura máxima do mês de Abril do ano de 1143 for superior a 23°, misture o conteúdo de duas embalagens em 5 litros de água, caso contrário, misture apenas o conteúdo de uma embalagem”, não seríamos capazes de determinar qual a temperatura máxima do mês de Abril de 1143 e, consequentemente, não seríamos capazes de executar esta instrução. Uma instrução como a segunda que acabamos de descrever não pode fazer parte de um algoritmo, pois não pode ser executada com uma quantidade de esforço finita, num intervalo de tempo finito.

3. Um algoritmo deve terminar. O algoritmo deve levar a uma situação em que o objectivo tenha sido atingido e não existam mais instruções para ser executadas. Consideremos o seguinte algoritmo para elevar a pressão de um pneu acima de 28 libras: “enquanto a pressão for inferior a 28 libras, continue a introduzir ar”. É evidente que, se o pneu estiver furado, o algoritmo anterior pode não terminar (dependendo do tamanho do furo) e, portanto, não o vamos classificar como algoritmo.

O conceito de algoritmo é fundamental em informática. Existem mesmo pessoas que consideram a informática como o estudo dos algoritmos: o estudo de máquinas para executar algoritmos, o estudo dos fundamentos dos algoritmos e a análise de algoritmos.

1.3

Programas e algoritmos

Um algoritmo, escrito de modo a poder ser executado por um computador, tem o nome de programa. Uma grande parte deste livro é dedicada ao desenvolvimento de algoritmos, e à sua codificação utilizando uma linguagem de programação, o Python. Os programas que desenvolvemos apresentam aspectos semelhantes aos algoritmos informais apresentados na secção anterior. Nesta secção, discutimos alguns desses aspectos.

Vimos que a receita dos rebuçados de ovos era constituída por uma descrição dos objectos a manipular (500 g de açúcar, 5 gemas de ovos) e uma descrição das acções a efectuar sobre esses objectos (leva-se o açúcar ao lume, deixa-se ferver até fazer ponto de pérola). A constituição de um programa é semelhante à de uma receita.

12

CAPÍTULO 1. COMPUTADORES, ALGORITMOS E PROGRAMAS

Num programa, existem entidades que são manipuladas pelo programa e existe uma descrição, numa linguagem apropriada, de um algoritmo que especifica as operações a realizar sobre essas entidades. Em algumas linguagens de programação, por exemplo, o C e o Java, todas as entidades manipuladas por um programa têm que ser descritas no início do programa, noutras linguagens, como é o caso do Python, isso não é necessário.

No caso das receitas de culinária, as entidades a manipular podem existir antes do início da execução do algoritmo (por exemplo, 500 g de açúcar) ou entidades que são criadas durante a sua execução (por exemplo, a massa de ovos).

A manipulação destas entidades vai originar um produto que é o objectivo do algoritmo (no nosso exemplo, os rebuçados de ovos). Analogamente, nos nossos programas, iremos manipular valores de variáveis. As variáveis vão-se compor-tar de um modo análogo aos ingredientes da receita dos rebuçados de ovos.

Tipicamente, o computador começa por receber certos valores para algumas das variáveis, após o que efectua operações sobre essas variáveis, possivelmente atribuindo valores a novas variáveis e, finalmente, chega a um conjunto de valores que constituem o resultado do programa.

As operações a efectuar sobre as entidades devem ser compreendidas pelo agente que executa o algoritmo. Essas acções devem ser suficientemente elementares para poderem ser executadas facilmente

pelo agente que executa o algoritmo. É

importante notar que, pelo facto de nos referirmos a estas acções como “acções elementares”, isto não significa que elas sejam operações atómicas (isto é, indecomponíveis). Elas podem referir-se a um conjunto de acções mais simples a serem executadas numa sequência bem definida.

1.3.1

Linguagens de programação

ao

Definimos uma linguagem de programação como uma linguagem utilizada para escrever programas de computador. Existem muitos tipos de linguagens de programação. De acordo com as afinidades que estas apresentam com o modo como os humanos resolvem problemas, podem ser classificadas em linguagens máquina, linguagens “assembly” e linguagens de alto nível.

A linguagem máquina é a linguagem utilizada para comandar directamente as acções do computador. As instruções em linguagem máquina são constituídas por uma sequência de dois símbolos discretos, correspondendo à existência ou à

1.3. PROGRAMAS E ALGORITMOS

13

ausência de sinal (normalmente representados por 0 e por 1, respectivamente) e manipulam directamente entidades dentro do computador. A linguagem máquina é difícil de usar e de compreender por humanos e varia de computador para computador (é a sua linguagem nativa). A linguagem “assembly”

é semelhante à linguagem máquina, diferindo desta no sentido em que usa nomes simbólicos com significado para humanos em lugar de sequências de zeros e de uns. Tal como a linguagem máquina, a linguagem “assembly” varia de computador para computador. As linguagens de alto nível aproximam-se das linguagens que os humanos usam para resolver problemas e, consequentemente, são muito mais fáceis de utilizar do que as linguagens máquina ou “assembly”, para além de poderem ser utilizadas em computadores diferentes. O Python é um exemplo de uma linguagem de alto nível.

Para que os computadores possam “entender” os programas escritos numa linguagem de alto nível (recorde-se que a linguagem máquina é a linguagem que o computador compreende), existem programas que traduzem as instruções de linguagens de alto nível em linguagem máquina, chamados processadores da linguagem. Existem fundamentalmente dois processos para fazer esta tradução, conhecidos por compilação e por interpretação. No caso do Python, isto é feito através de um programa chamado o interpretador, que recebe instruções em Python e que é capaz de executar as acções correspondentes a cada uma delas.

1.3.2

Exemplo de um programa

Apresentamos um exemplo de algoritmo para calcular a soma dos 100 primeiros inteiros positivos e o programa correspondente em Python. Para isso, vamos analisar o nosso comportamento ao resolver este problema utilizando uma calculadora. O objectivo da nossa apresentação é fornecer uma ideia intuitiva dos passos e do raciocínio envolvidos na geração de um programa e, simultaneamente, dar uma primeira ideia de um programa em Python. Podemos descrever as acções a executar para resolver este problema através da seguinte sequência de comandos a fornecer à calculadora:

Limpar o visor da calculadora

Carregar na tecla 1

Carregar na tecla +

Carregar na tecla 2

14

CAPÍTULO 1. COMPUTADORES, ALGORITMOS E PROGRAMAS

Carregar na tecla +

Carregar na tecla 3

Carregar na tecla +

...

Carregar na tecla 1

Carregar na tecla 0

Carregar na tecla 0

Carregar na tecla =

Os símbolos “...” na nossa descrição de acções indicam que existe um padrão que se repete ao longo da nossa actuação e portanto não é necessário enumerar todos os passos, porque podemos facilmente gerar e executar os que estão implícitos.

No entanto, a existência destes símbolos não permite qualificar o conjunto de instruções anteriores como “algoritmo”, pois a característica da precisão deixa de se verificar. Para executar este conjunto de instruções é necessário ter a capacidade de compreender quais são os passos subentendidos por “...”.

Para transformar as instruções anteriores num algoritmo que possa ser executado por um computador, é necessário tornar explícito o que está implícito.

Note-se, no entanto, que para explicitar todos os passos do algoritmo anterior teríamos mais trabalho do que se executássemos o algoritmo nós próprios, pelo que será conveniente encontrarmos uma formulação alternativa. Para isso, vamos reflectir sobre o processo de cálculo aqui descrito. Existem duas grandezas envolvidas neste processo de cálculo, a soma corrente (que aparece, em cada instante, no visor da calculadora) e o número a ser adicionado à soma (o qual é mantido na nossa cabeça). Cada vez que um número é adicionado à soma corrente, mentalmente, aumentamos em uma unidade o próximo número a ser adicionado. Se quisermos exprimir este processo de um modo rigoroso, necessitamos de recorrer a duas variáveis, uma para representar a soma corrente (à qual chamaremos soma), e a outra para representar o número que mantemos na nossa cabeça (a que chamaremos numero). Os passos repetitivos que executamos são:

A soma toma o valor de soma + numero

O numero toma o valor de numero + 1

O ciclo que executamos ao calcular a soma dos 100 primeiros inteiros positivos é:

1.3. PROGRAMAS E ALGORITMOS

enquanto o numero for menor ou igual a 100

A soma toma o valor de soma + numero

O numero toma o valor de numero + 1

Convém agora relembrar as operações que efectuamos antes de começar a executar esta sequência repetitiva de operações: (1) limpámos o visor da calculadora, isto é estabelecemos que o valor inicial da variável soma era zero; (2) estabelecemos que o primeiro numero a ser adicionado à soma era um. Com estes dois aspectos em mente, poderemos dizer que a sequência de passos a seguir para calcular a soma dos 100 primeiros inteiros positivos é: A soma toma o valor de 0

O numero toma o valor de 1

enquanto o numero for menor ou igual a 100

A soma toma o valor de soma + numero

O numero toma o valor de numero + 1

Em matemática, operações como “toma o valor de” são normalmente representadas por um símbolo (por exemplo, =). Em programação, esta operação é também representada por um símbolo (=, :=, etc., dependendo da linguagem de programação utilizada). Se adoptarmos o símbolo utilizado em Python, =, o nosso algoritmo será representado por:

```
soma = 0
```

```
numero = 1
```

```
enquanto numero ≤ 100
```

```
soma = soma + numero
```

```
numero = numero + 1
```

Esta descrição é uma versão muito aproximada de um programa em Python para calcular a soma dos primeiros 100 inteiros positivos, o qual é o seguinte: `def prog_soma () :`

```
soma = 0
```

```
numero = 1
```

```
while numero <= 100 :
```

```
soma = soma + numero
```

```
numero = numero + 1
```

```
print('O valor da soma é: ', soma)
```

16

CAPÍTULO 1. COMPUTADORES, ALGORITMOS E PROGRAMAS

É importante notar que existe uma fórmula que nos permite calcular a soma dos primeiros 100 inteiros positivos, se os considerarmos como uma progressão aritmética de razão um, a qual é dada por:

$$100 \cdot (1 + 100)$$

soma =

2

e, consequentemente, poderíamos utilizar esta fórmula para obter o valor desejado da soma, o que poderá ser representado pelo seguinte programa em Python5:

```
def prog_soma () :
```

```
soma = (100 * (1 + 100)) // 2
```

```
print('O valor da soma é: ', soma)
```

Um aspecto importante a reter a propósito deste exemplo é o facto de normalmente não existir apenas um algoritmo (e consequentemente apenas um programa) para resolver um dado problema. Estes algoritmos podem ser muito diferentes entre si.

1.4

Sintaxe e semântica

O Python, como qualquer linguagem, apresenta dois aspectos distintos: as frases da linguagem e o significado associado às frases. Estes aspectos são denominados, respectivamente, sintaxe e semântica da linguagem.

Ao estudar uma linguagem de programação, é fundamental uma

perfeita compreensão da sua sintaxe e da sua semântica: a sintaxe determina qual a constituição das frases que podem ser fornecidas ao computador, e a semântica vai determinar o que o computador vai fazer ao seguir cada uma dessas frases.

1.4.1

Sintaxe

A sintaxe de uma linguagem é o conjunto de regras que definem quais as relações válidas entre os componentes da linguagem, tais como as palavras e as frases.

A sintaxe nada diz em relação ao significado das frases da linguagem.

Em Python, “*” representa a multiplicação e “//” representa a divisão inteira.

1.4. SINTAXE E SEMÂNTICA

17

Em linguagem natural, a sintaxe é conhecida como a gramática. Analogamente, em linguagens de programação, a sintaxe também é definida através de gramáticas. Como exemplo da gramática de uma linguagem natural, vamos considerar um fragmento do português. Em português, uma frase simples é constituída por um sintagma nominal seguido de um sintagma verbal (um sintagma é uma subdivisão de uma frase). Por exemplo, a frase “O João acendeu o cachimbo” é constituída pelo sintagma nominal “O João” e pelo sintagma verbal “acendeu o cachimbo”. Uma das possíveis constituições de um sintagma nominal consiste num determinante (por exemplo, um artigo) seguido de um nome. No exemplo apresentado, o sintagma nominal “O João” é constituído pelo determinante “O”, seguido do nome “João”. Um sintagma verbal pode ser definido como sendo um verbo seguido de um sintagma nominal. No nosso exemplo, o sintagma verbal “acendeu o cachimbo” é constituído pelo verbo

“acendeu” e pelo sintagma nominal “o cachimbo”. De acordo com o que temos vindo a apresentar sobre a estrutura de uma frase em português, a frase “O

cachimbo acendeu o João” é uma frase sintacticamente correcta, embora sem qualquer significado. Isto apoia a ideia de que a sintaxe não se preocupa com o significado associado às frases, mas apenas com a definição da sua estrutura.

Como a sintaxe apenas se preocupa com o processo de combinação dos símbolos de uma linguagem, ela pode ser, na maior parte dos casos, facilmente formalizada. Os linguistas e os matemáticos estudaram as propriedades sintáticas das linguagens, e grande parte deste trabalho é aplicável às linguagens de programação.

Para descrever a sintaxe do Python, utilizaremos uma notação conhecida por notação BNF, a qual utiliza as seguintes regras: 1. Para designar um componente da linguagem escrevemo-lo entre parênteses angulares, “ $\langle \rangle$ ” e “ $[\]$ ”. Por exemplo, $\langle \text{expressão} \rangle$ representa uma expressão em Python. Não estamos, aqui, a falar de nenhuma expressão em particular, mas sim a falar de um componente genérico da linguagem. Por esta razão, os nomes que aparecem entre “ $\langle \rangle$ ” e “ $[\]$ ” são designados por símbolos não terminais. Um símbolo não terminal está sempre associado a um conjunto de entidades da linguagem, no nosso exemplo, todas as expressões.

2. Ao descrever a linguagem, teremos também de falar das entidades que aparecem explicitamente nas frases da linguagem, por exemplo o símbolo

18

CAPÍTULO 1. COMPUTADORES, ALGORITMOS E PROGRAMAS

“+”. Os símbolos que aparecem nas frases da linguagem são chamados símbolos terminais e são escritos, em notação BNF, sem qualquer símbolo especial à sua volta.

3. O símbolo “ ” (lido “ou”) representa possíveis alternativas.

4. O símbolo “::=” (lido “é definido como”) serve para definir componentes da linguagem.

5. A utilização do carácter “+” imediatamente após um símbolo não terminal significa que esse símbolo pode ser repetido uma ou mais vezes.

6. A utilização do carácter “*” imediatamente após um símbolo não terminal significa que esse símbolo pode ser repetido zero ou mais vezes.

7. A utilização de chavetas, “{” e “}”, englobando símbolos terminais ou não terminais, significa que esses símbolos são opcionais.

Para aumentar a facilidade de leitura das nossas gramáticas, vamos

usar dois tipos de letra, respectivamente para os símbolos terminais e para os símbolos não terminais: os símbolos terminais são escritos utilizando uma letra correspondente ao tipo máquina de escrever (como é feito nas palavras “símbolos terminais”, em cima); os símbolos não terminais são escritos usando um tipo helvética (como é feito nas palavras “símbolos não terminais”, em cima). Note-se, contudo, que esta convenção apenas serve para aumentar a facilidade de leitura das expressões e não tem nada a ver com as propriedades formais das nossas gramáticas.

Como primeiro exemplo, vamos considerar uma gramática em notação BNF para o subconjunto de português descrito no início desta secção. Dissemos que uma frase é constituída por um sintagma nominal seguido de um sintagma verbal, utilizando $\Diamond F \Diamond$, $\Diamond SN \Diamond$ e $\Diamond SV \Diamond$ para representar, respectivamente, uma frase, um sintagma nominal e um sintagma verbal. A regra da gramática em notação BNF

que define a estrutura de uma frase é

$$\Diamond F \Diamond ::= \Diamond SN \Diamond \Diamond SV \Diamond$$

Esta regra é lida do seguinte modo: uma frase é definida como (“é definida como” é representado pelo símbolo “::=”) um sintagma nominal seguido por um sintagma verbal (“seguido por” está implícito quando usamos um símbolo não terminal seguido de outro símbolo não terminal).

1.4. SINTAXE E SEMÂNTICA

19

F

SN

SV

Det

N

V

SN

Det

N

O

João

acendeu

o

cachimbo

Figura 1.3: Estrutura da frase “O João acendeu o cachimbo”.

Um sintagma nominal é um determinante seguido de um nome. Isto pode ser expresso através da seguinte regra em notação BNF:

$$\langle \text{SN} \rangle ::= \langle \text{Det} \rangle \langle \text{N} \rangle$$

em que $\langle \text{Det} \rangle$ representa um determinante e $\langle \text{N} \rangle$ representa um nome. Finalmente, um sintagma verbal é definido como um verbo seguido de um sintagma nominal, o que pode ser escrito através da seguinte expressão em notação BNF:

$$\langle \text{SV} \rangle ::= \langle \text{V} \rangle \langle \text{SN} \rangle$$

em que $\langle \text{V} \rangle$ representa um verbo.

Podemos ainda dizer que o é um determinante, João e cachimbo são nomes e que acendeu é um verbo:

$$\langle \text{Det} \rangle ::= \text{o}$$
$$\langle \text{N} \rangle ::= \text{João} \mid \text{cachimbo}$$
$$\langle \text{V} \rangle ::= \text{acendeu}$$

A segunda regra é lida do seguinte modo: um nome é definido como João ou cachimbo, em que “ou” corresponde ao símbolo “ ”.

Note-se que existe uma relação directa entre as regras da gramática em notação BNF e a estrutura das frases da linguagem. Na Figura 1.3 apresentamos a estrutura da frase “O João acendeu o cachimbo”. De facto, cada regra da gramática pode ser considerada como definindo uma estrutura em que o símbolo à sua esquerda está imediatamente acima dos símbolos à sua direita (dizemos que o

CAPÍTULO 1. COMPUTADORES, ALGORITMOS E PROGRAMAS

símbolo à sua esquerda domina os símbolos à sua direita). Na Figura 1.3, $\Diamond F \Diamond$

domina tanto $\Diamond SN \Diamond$ como $\Diamond SV \Diamond$, correspondendo à regra $\Diamond F \Diamond ::= \Diamond SN \Diamond \Diamond SV \Diamond; \Diamond SN \Diamond$

domina tanto $\Diamond Det \Diamond$ como $\Diamond N \Diamond$, correspondendo à regra $\Diamond SN \Diamond ::= \Diamond Det \Diamond \Diamond N \Diamond$, e assim sucessivamente. Nesta figura não representamos os parênteses angulares nos símbolos não terminais, pois a distinção entre símbolos terminais e símbolos não terminais é óbvia, os símbolos terminais não dominam quaisquer outros símbolos.

Como segundo exemplo, consideremos uma gramática em notação BNF para definir números binários. Informalmente, um número binário é apenas constituído pelos dígitos binários 0 e 1, podendo apresentar qualquer quantidade destes dígitos ou qualquer combinação entre eles. A seguinte gramática define números binários:

$\Diamond \text{número binário} \Diamond ::= \Diamond \text{dígito binário} \Diamond$

$\Diamond \text{dígito binário} \Diamond \Diamond \text{número binário} \Diamond$

$\Diamond \text{dígito binário} \Diamond ::= 0 \ 1$

Nesta gramática os símbolos terminais são 0 e 1 e os símbolos não terminais são

$\Diamond \text{número binário} \Diamond$ e $\Diamond \text{dígito binário} \Diamond$. A gramática tem duas regras. A primeira define a classe dos números binários, representados pelo símbolo não terminal

$\Diamond \text{número binário} \Diamond$, como sendo um $\Diamond \text{dígito binário} \Diamond$, ou um $\Diamond \text{dígito binário} \Diamond$ seguido de um $\Diamond \text{número binário} \Diamond$. A segunda parte desta regra diz simplesmente que um número binário é constituído por um dígito binário seguido por um número binário. Sucessivas aplicações desta regra levam-nos a concluir que um número binário pode ter tantos dígitos binários quantos queiramos (ou seja, podemos aplicar esta regra tantas vezes quantas desejarmos). Podemos agora perguntar quando é que paramos a sua aplicação. Note-se que a primeira parte desta mesma regra diz que um número binário é um dígito binário. Portanto, sempre que utilizamos a primeira parte desta

regra, terminamos a sua aplicação. A segunda regra de produção define um dígito binário, representado pelo símbolo não terminal dígito binário, como sendo ou 0 ou 1. Na Figura 1.4 mostramos a estrutura do número binário 101 de acordo com a gramática apresentada.

Como último exemplo, consideremos a seguinte gramática que define um maior subconjunto do português do que o apresentado no primeiro exemplo (nesta 6 É importante compreender bem esta regra. Ela representa o primeiro contacto com uma classe muito importante de definições

as chamadas definições

as recursivas (ou definições

as por recorrência), nas quais uma entidade é definida em termos de si própria. As definições

as recursivas

são

as discutidas em pormenor no Capítulo 6.

1.4. SINTAXE E SEMÂNTICA

21

número binário

dígito binário

número binário

dígito binário

número binário

dígito binário

1

0

1

Figura 1.4: Estrutura do número binário 101.

gramática, $\Diamond SP \Diamond$ representa um sintagma preposicional e $\Diamond Prep \Diamond$ representa uma preposição):

$\Diamond F \Diamond ::= \Diamond SN \Diamond \Diamond SV \Diamond \Diamond SN \Diamond \Diamond SV \Diamond \Diamond SP \Diamond$

$\Diamond SN \Diamond ::= \Diamond N \Diamond \Diamond Det \Diamond \Diamond N \Diamond \Diamond N \Diamond \Diamond SP \Diamond \Diamond Det \Diamond \Diamond N \Diamond \Diamond SP \Diamond$

$\Diamond SV \Diamond ::= \Diamond V \Diamond \Diamond V \Diamond \Diamond SN \Diamond \Diamond V \Diamond \Diamond SP \Diamond \Diamond V \Diamond \Diamond SN \Diamond \Diamond SP \Diamond$

$\Diamond SP \Diamond ::= \Diamond Prep \Diamond \Diamond SN \Diamond$

$\Diamond Det \Diamond ::= O \text{ um}$

$\Diamond N \Diamond ::= Eu \text{ homem telescópio monte}$

$\Diamond V \Diamond ::= vi$

$\Diamond Prep \Diamond ::= em \text{ com}$

É importante dizer que uma gramática completa para o português é extremamente complicada e não pode ser formalizada utilizando apenas a notação BNF.

A gramática que apresentámos permite a construção de frases como “Eu vi o homem no monte com um telescópio”. Notemos que esta frase pode ser gerada, pelo menos, de dois modos diferentes (as estruturas correspondentes estão representadas nas figuras 1.5 e 1.6). Nestas estruturas, as palavras “em o” são contraídas para gerar a palavra “no”.

Uma gramática que pode originar a mesma frase com estruturas diferentes é denominada ambígua. As gramáticas para as linguagens naturais são inerentemente ambíguas. Um problema associado às gramáticas ambíguas é que às

N

V

SN

SP

Det

N

SP

Prep

SN

Prep

SN

Det

N

Det

N

Eu

vi

o

homem

em

o

monte

com

o

telescópio

Figura 1.5: Possível estrutura da frase “Eu vi o homem no monte com um telescópio”.

F

SV

SN

N

V

SN

Det

N

SP

Prep

SN

SP

Det

N

Prep

SN

Det

N

Eu

vi

o

homem

em
o
monte
com
o
telescópio

Figura 1.6: Estrutura alternativa da frase “Eu vi o homem no monte com um telescópio”.

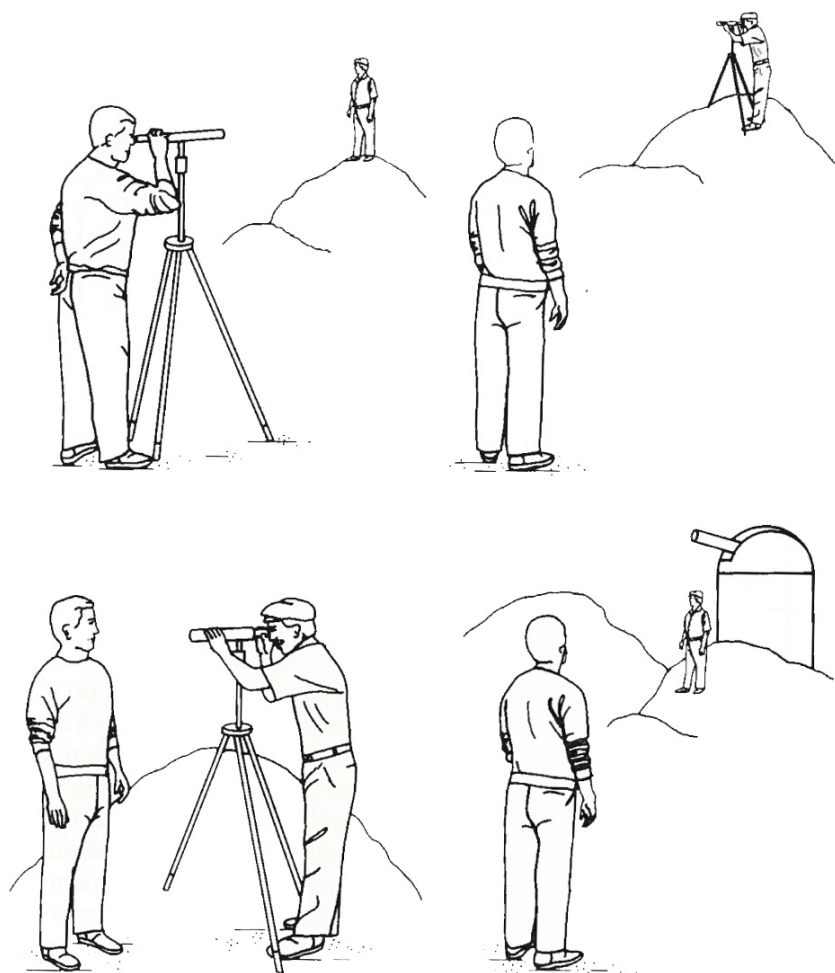


Figura 1.7: Possíveis significados da frase “Eu vi o homem no monte com um telescópio”.

diferentes estruturas da mesma frase estão associados significados diferentes, e portanto, dada uma frase isolada, não é possível determinar univocamente o seu significado. Na Figura 1.7 7 apresentamos alguns dos possíveis significados da frase “Eu vi o homem no monte com um telescópio”. Em linguagem natural, a ambiguidade das frases é normalmente resolvida tendo em atenção o contexto da conversa em que a frase aparece. Por exemplo, a frase “Eu vi o homem no monte com um telescópio”, seguida de uma das frases “Limpei a lente para o ver melhor” ou “Falámos sobre astronomia”, selecciona uma das possíveis interpretações da Figura 1.7 (ver a Figura 1.8 8).

Nas linguagens de programação, a ambiguidade sintáctica não pode existir. Ao analisar uma frase de uma linguagem de programação, o computador deve ser capaz de determinar exactamente a estrutura dessa frase e, consequentemente, o seu significado. Uma das preocupações na definição de uma linguagem de 7Adaptada de [Lenhert and Ringle, 1982], página 21, Figura 1.5. Reproduzida com autorização

ao de Lawrence Erlbaum Associates, Inc. e dos autores.

8ibid.

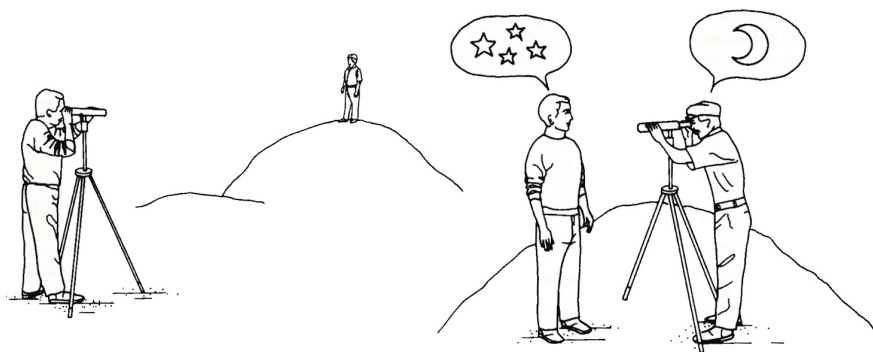


Figura 1.8: Resolução de ambiguidade na frase “Eu vi o homem no monte com um telescópio”.

programação é a de evitar qualquer tipo de ambiguidade.

A notação utilizada para definir formalmente uma linguagem, no caso da notação BNF, “ $\diamond$ ”, “ $\diamond$ ”, “ ”, “ $::=$ ”, “{”, “}”, “+”, “*”, os símbolos não terminais e os símbolos terminais, é denominada metalinguagem, visto ser a linguagem que utilizamos para falar acerca de outra linguagem (ou a linguagem que está para além da linguagem). Um dos poderes da formalização da sintaxe utilizando metalinguagem é tornar perfeitamente clara a distinção entre “falar acerca da linguagem” e “falar com a linguagem”. A confusão entre linguagem e metalinguagem pode levar a paradoxos de que é exemplo a frase “esta frase é falsa”.

1.4.2

Semântica

“Then you should say what you mean,” the March Hare went on.

“I do,” Alice hastily replied; “at least—at least I mean what I say—that’s the same thing, you know.”

“Not the same thing a bit!” said the Hatter. “You might just as well say that “I see what I eat” is the same thing as “I eat what I see”!”

Lewis Carroll, Alice’s Adventures in Wonderland A semântica de uma linguagem define qual o significado de cada frase da linguagem. A semântica nada diz quanto ao processo de geração das frases da linguagem. A descrição da semântica de uma linguagem de programação é muito mais difícil do que a descrição da sua sintaxe. Um dos processos de des-

1.4. SINTAXE E SEMÂNTICA

crever a semântica de uma linguagem consiste em fornecer uma descrição em língua natural (por exemplo, em português) do significado, ou seja, das acções que são realizadas pelo computador, de cada um dos possíveis componentes da linguagem. Este processo, embora tenha os inconvenientes da informalidade e da ambiguidade

associadas às línguas naturais, tem a vantagem de fornecer uma perspectiva intuitiva da linguagem.

Cada frase em Python tem uma semântica, a qual corresponde às acções tomadas pelo Python ao executar essa frase, ou seja, o significado que o Python atribui à frase. Esta semântica é definida por regras para extrair o significado de cada frase, as quais são descritas neste livro de um modo incremental, à medida que novas frases são apresentadas. Utilizaremos o português para exprimir a semântica do Python.

1.4.3

Tipos de erros num programa

De acordo com o que dissemos sobre a sintaxe e a semântica de uma linguagem, deverá ser evidente que um programa pode apresentar dois tipos distintos de erros: erros de natureza sintáctica e erros de natureza semântica.

Os erros de natureza sintáctica, ou erros sintácticos resultam do facto de o programador não ter escrito as frases do seu programa de acordo com as regras da gramática da linguagem de programação utilizada. A detecção destes erros é feita pelo processador da linguagem, o qual fornece normalmente um diagnóstico sobre o que provavelmente está errado. Todos os erros de natureza sintáctica têm que ser corrigidos antes da execução das instruções, ou seja, o computador não executará nenhuma instrução sintacticamente incorrecta. Os programadores novatos passam grande parte do seu tempo a corrigir erros sintácticos, mas à medida que se tornam mais experientes, o número de erros sintácticos que originam é cada vez menor e a sua origem é detectada de um modo cada vez mais rápido.

Os erros de natureza semântica, ou erros semânticos (também conhecidos por erros de lógica) são erros em geral muito mais difíceis de detectar do que os erros de carácter sintáctico. Estes erros resultam do facto de o programador não ter expressado correctamente, através da linguagem de programação, as acções a serem executadas (o programador queria dizer uma coisa mas disse outra).

Os erros sintácticos podem-se manifestar pela geração de uma mensagem de

CAPÍTULO 1. COMPUTADORES, ALGORITMOS E PROGRAMAS

erro durante a execução de um programa, pela produção de resultados errados ou pela geração de ciclos que nunca terminam. Neste livro apresentaremos técnicas de programação que permitem minimizar os erros semânticos e, além disso, discutiremos métodos a utilizar para a detecção e correção dos erros de natureza semântica de um programa.

Ao processo de detecção e correção, tanto dos erros sintáticos como dos erros semânticos, dá-se o nome de depuração (do verbo depurar, tornar puro, limpar).

Em inglês, este processo é denominado “debugging” e aos erros que existem num programa, tanto sintáticos como semânticos, chamam-se “bugs”⁹. O termo

“bug” foi criado pela pioneira da informática Grace Murray Hopper (1906–

1992). Em Agosto de 1945, Hopper e alguns dos seus associados estavam a trabalhar em Harvard com um computador experimental, o Mark I, quando um dos circuitos deixou de funcionar. Um dos investigadores localizou o problema e, com auxílio de uma pinça, removeu-o: uma traça com cerca de 5 cm. Hopper colou a traça, com fita gomada, no seu livro de notas e disse: “A partir de agora, sempre que um computador tiver problemas direi que ele contém insectos (bugs)”. A traça ainda hoje existe, juntamente com os registos das experiências, no “U.S. Naval Surface Weapons Center” em Dahlgren, Virginia, nos Estados Unidos da América¹⁰.

Para desenvolver programas, são necessárias duas competências fundamentais, a capacidade de resolução de problemas que corresponde à competência para formular o problema que deve ser resolvido pelo programa, criar uma solução para esse problema, através da sua divisão em vários subproblemas mais simples, e expressar essa solução de um modo rigoroso recorrendo a uma linguagem de programação e a capacidade de depuração que consiste em, através de uma análise rigorosa, perceber quais os erros existentes no programa e corrigi-los adequadamente. A depuração é fundamentalmente um trabalho de detective em que se analisa de uma forma sistemática o que está a ser feito pelo programa, formulando hipóteses sobre o que está mal e testando essas hipóteses através da modificação do programa. A depuração semântica é frequentemente uma tarefa difícil, requerendo espírito crítico e persistência.

9Do inglês, insectos.

10Ver [Taylor, 1984], página 44.

1.5. NOTAS FINAIS

27

1.5

Notas finais

Neste capítulo apresentámos alguns conceitos básicos em relação à programação.

Um computador, como uma máquina cuja função é a manipulação de símbolos, e as suas características fundamentais, ser automático, universal, electrónico e digital. Uma apresentação informal muito interessante sobre as origens dos computadores e dos matemáticos ligados à sua evolução pode ser consultada em

[Davis, 2004].

Introduzimos a noção de programa, uma sequência de instruções escritas numa linguagem de programação, e o resultado originado pela execução de um programa, um processo computacional.

Apresentámos o conceito de algoritmo, uma sequência finita de instruções bem definidas e não ambíguas, cada uma das quais pode ser executada mecanicamente num período de tempo finito com uma quantidade de esforço finita, bem como as suas características, ser rigoroso, eficaz e dever terminar. O aspecto de um algoritmo ter que terminar é de certo modo controverso. Alguns autores, por exemplo [Hennie, 1977] e [Hermes, 1969], admitem que um algoritmo possa não terminar. Para estes autores, um algoritmo apenas apresenta as características de rigor e de eficácia. Outros autores, por exemplo [Brainerd and Landwebber, 1974] e [Hopcroft and Ullman, 1969], distinguem entre procedimento mecânico

– uma sequência finita de instruções que pode ser executada mecanicamente –

e um algoritmo – um procedimento mecânico que é garantido terminar. Neste livro, adoptamos a segunda posição.

Finalmente apresentámos os dois aspectos associados a uma

linguagem, a sintaxe e a semântica, e introduzimos a notação BNF para definir a sintaxe de uma linguagem. Como a sintaxe apenas se preocupa com o processo de combinação dos símbolos de uma dada linguagem, ela pode ser, na maior parte dos casos, facilmente formalizada. Os linguistas e os matemáticos estudaram as propriedades sintáticas das linguagens, e grande parte deste trabalho é aplicável às linguagens de programação. É particularmente importante o trabalho de Noam Chomsky ([Chomsky, 1957] e [Chomsky, 1959]), que classifica as linguagens em grupos. Grande parte das linguagens de programação pertence ao grupo 2, ou grupo das linguagens livres de contexto (do inglês, “context-free languages”).

A notação BNF foi inventada por John Backus e Peter Naur, e a sua primeira utilização significativa foi na definição da sintaxe da linguagem Algol 60. O

28

CAPÍTULO 1. COMPUTADORES, ALGORITMOS E PROGRAMAS

termo BNF significa “Backus-Naur Form”. Alguns autores, por exemplo [Hopcroft and Ullman, 1969] e [Ginsburg, 1966] usam BNF com o significado de

“Backus Normal Form”.

1.6

Exercícios

1. Escreva uma gramática em notação BNF para definir um número inteiro.

Um número inteiro é um número, com ou sem sinal, constituído por um número arbitrário de dígitos.

2. Escreva uma gramática em notação BNF para definir um número real, o qual pode ser escrito quer em notação decimal quer em notação científica.

Um real em notação decimal pode ou não ter sinal, e tem que ter ponto decimal, o qual é rodeado por dígitos. Por exemplo, +4.0, -4.0 e 4.0

são números reais em notação decimal. Um real em notação científica tem uma mantissa, a qual é um inteiro ou um real, o símbolo

“e” é um expoente inteiro, o qual pode ou não ter sinal. Por exemplo, $4.2e-5$, $2e4$

e $-24.24e+24$ são números reais em notação científica.

3. Considere a seguinte gramática em notação BNF’:

$$\langle S \rangle ::= \langle A \rangle a$$

$$\langle A \rangle ::= a \langle B \rangle$$

$$\langle B \rangle ::= \langle A \rangle a b$$

(a) Diga quais são os símbolos terminais e quais são os símbolos não terminais da gramática.

(b) Quais das frases pertencem ou não à linguagem definida pela gramática.

Justifique a sua resposta.

aabaa

abc

abaa

aaaabaaaa

4. Considere a seguinte gramática em notação BNF:

$$\langle \text{Comp} \rangle ::= \langle \text{Bum}^* \rangle \langle \text{Dum} \rangle + \langle \text{Bum}^* \rangle$$

$$\langle \text{Bum}^* \rangle ::= \langle \text{Bum} \rangle \langle \text{Vazio} \rangle$$

1.6. EXERCÍCIOS

29

$$\langle \text{Dum} \rangle + \langle \rangle ::= \langle \text{Dum} \rangle \langle \text{Dum} \rangle + \langle \text{Dum} \rangle$$

$$\langle \text{Dum} \rangle ::= B C D$$

$$\langle \text{Bum} \rangle ::= A E$$

$$\langle \text{Vazio} \rangle ::=$$

(a) Diga quais são os símbolos terminais e quais são os símbolos

não terminais da gramática.

(b) Quais das seguintes frases pertencem à linguagem definida pela gramática? Justifique a sua resposta.

EFA

AA

ACC

DIB

BBAAAAAE

5. Escreva uma gramática em notação BNF que defina frases da seguinte forma: (1) as frases começam por c; (2) as frases acabam em r; (3) entre o c e o r podem existir tantos a's e d's quantos quisermos, mas tem que existir pelo menos um deles. São exemplos de frases desta linguagem: car, cadar, cdr e cddddr.

6. Considere a representação de tempo utilizada em relógios digitais, na qual aparecem as horas (entre 0 e 23), minutos e segundos. Por exemplo 10:23:45.

(a) Descreva esta representação utilizando uma gramática em notação BNF.

(b) Quais são os símbolos terminais e quais são os símbolos não terminais da sua gramática?

7. Considere a seguinte gramática em notação BNF:

$\text{Blop} ::= c \text{ Gulp } s$

$\text{Gulp} ::= a b \text{ Gulp } x \text{ Bat } y$

$\text{Bat} ::= x \text{ Bat } y$

(a) Diga quais são os símbolos terminais e quais são os símbolos não terminais da gramática.

30

CAPÍTULO 1. COMPUTADORES, ALGORITMOS E PROGRAMAS

(b) Para cada uma das seguintes frases, diga se ela pertence ou não à

linguagem definida pela gramática. Se não pertence, indique o primeiro símbolo que impossibilita a pertença.

cas

cGulps

cxxyyys

cxxyys

abxy

bay

cass

cbbbbbbxxy

cxxy

x

8. Dada a seguinte gramática em notação BNF:

$S ::= b B$

$B ::= b C \mid a B \mid b$

$C ::= a$

(a) Diga quais os símbolos terminais e quais são os símbolos não terminais desta gramática.

(b) Diga, justificando, se as seguintes frases pertencerem ou não à linguagem definida pela gramática:

baaab

aabb

bba

baaaaaaba

9. A notação BNF é um formalismo sintático e, como tal, pode ser definido utilizando uma notação do tipo BNF. Escreva uma gramática do tipo BNF

para descrever a sintaxe das gramáticas do tipo BNF. Sugest~ao: vai ser necessário introduzir uma nova notaç~ao.

Capítulo 2

Elementos básicos de

programaç~

ao

The White Rabbit put on his spectacles.

‘Where shall I begin, please your Majesty?’ he asked.

‘Begin at the beginning,’ the king said, very gravely, ‘and go on till you come to the end: then stop.’

Lewis Carroll, Alice’s Adventures in Wonderland No capítulo anterior, vimos que um programa corresponde a um algoritmo escrito numa linguagem de programaç~ao. Dissemos também que qualquer programa é num pequeno número de operaç~oes elementares, entre as quais se encontram as operaç~oes de entrada de dados, as operaç~oes de saída de dados, operaç~oes matemáticas, a execuç~ao condicional e a repetiç~ao.

Neste capítulo apresentamos algumas noç~oes básicas associadas a um programa, nomeadamente, algumas das operaç~oes correspondentes às operaç~oes elementares que acabámos de listar. No final deste capítulo, estaremos em condiç~oes de escrever alguns programas muito simples.

Um programa manipula variáveis usando um conjunto de operaç~oes. Apresentamos neste capítulo alguns dos tipos de valores que podem ser associados a variáveis, o modo de os combinar, juntamente com algumas instruções existentes em Python.

O Python é uma linguagem de programaç~ao, ou seja, corresponde a um forma-31

32

CAPÍTULO 2. ELEMENTOS B ÁSICOS DE PROGRAMAC

~
,

AO

lismo para escrever programas. Um programa em Python pode ser introduzido e executado interactivamente num ambiente em que exista um interpretador do Python – uma “caixa electrónica” que compreenda as frases da linguagem Python.

A interacção entre um utilizador e o Python é feita através de um teclado e de um écran. O utilizador escreve frases através do teclado (aparecendo estas também no écran), e o computador responde, mostrando no écran do resultado de efectuar as acções indicadas na frase. Após efectuar as acções indicadas na frase, o utilizador fornece ao computador outra frase, e este ciclo repete-se até o utilizador terminar o trabalho. A este modo de interacção dá-se o nome de processamento interactivo. Em processamento interactivo, o utilizador dialoga com o computador, fornecendo uma frase de cada vez e esperando pela resposta do computador, antes de fornecer a próxima frase.

Ao iniciar uma sessão com o Python recebemos uma mensagem semelhante à seguinte:

```
Python 3.2.2 (v3.2.2:137e45f15c0b, Sep
```

```
3 2011, 16:48:10)
```

```
[GCC 4.0.1 (Apple Inc. build 5493)]
```

```
Type "help", "copyright", "credits" or "license" for more information.
```

```
> > >
```

O símbolo “> > >” é uma indicação de que o Python está pronto para receber uma frase. Este símbolo é chamado o carácter de pronto¹. A utilização interactiva do Python corresponde à repetição de um ciclo em que o Python lê uma frase, efectua as acções indicadas na frase e escreve o resultado dessas acções. Este ciclo é chamado ciclo lê-avalia-escreve². Uma sessão em Python corresponde a um ciclo, tão longo quanto o utilizador desejar, de leitura de frases, execução das acções indicadas na frase e apresentação dos resultados.

Ao longo do livro apresentamos exemplos de interacções com o Python. Ao apresentar uma interacção, o que aparece depois do símbolo “> > >” corresponde à informação que é fornecida ao Python, e o que aparece na linha, ou linhas, seguintes corresponde à resposta que é fornecida pelo Python.

Uma frase em Python é designada por um comando. Um comando pode ser uma 1Do inglês, “prompt character”.

2Do inglês “read-eval-print loop”.

2.1. EXPRESS ~

OES

33

express~ao, uma instruç~ao ou uma definiç~ao. Em notaç~ao BNF, um comando é definido do seguinte modo:

◆comando◆ ::= ◆express~ao◆ ◆instruç~ao◆ ◆definiç~ao◆

Começamos por analisar as express~oes em Python, após o que consideramos algumas instruç~oes elementares. O conceito de ◆definiç~ao◆ é introduzido no próximo capítulo.

2.1

Express~

oes

Um dos tipos de entidades que utilizamos nos nossos programas corresponde a express~oes. Por definiç~ao, uma express~ao é uma entidade computacional que tem um valor. Usamos o termo entidade computacional para designar, de um modo genérico, uma entidade que existe dentro de um programa.

Uma express~ao em Python pode ser uma constante, uma express~ao composta ou um nome. Em notaç~ao BNF, uma express~ao é definida do seguinte modo:

◆express~ao◆ ::= ◆constante◆ ◆express~ao composta◆ ◆nome◆

Nesta secç~ao, vamos considerar express~oes correspondentes a constantes e express~oes compostas utilizando algumas operaç~oes básicas.

2.1.1

Constantes

Para os efeitos deste capítulo, consideramos que as constantes em

Python podem ser números, valores lógicos ou cadeias de caracteres. Sempre que é fornecida uma constante ao Python, este devolve a constante como resultado da avaliação. Ou seja, o valor de uma constante é a própria constante (o Python mostra a representação externa da constante). A representação externa de uma entidade corresponde ao modo como nós visualizamos essa entidade, independentemente do modo como esta é representada internamente no computador (a representação interna), a qual, como sabemos, é feita recorrendo apenas aos símbolos 0 e 1. Por exemplo, a representação externa do inteiro 1958 é 1958 ao passo que a sua representação interna é 11110100110.

A seguinte interação mostra a resposta do Python quando lhe fornecemos algumas constantes:

34

CAPÍTULO 2. ELEMENTOS BÁSICOS DE PROGRAMAÇÃO

~
,

AO

> > > 1958

1958

> > > -1

-1

> > > +2

2

> > >

655484877641416186376754588877162243232221200091999228887333

655484877641416186376754588877162243232221200091999228887333

> > > 3.5

3.5

> > > 65397518517617656541959.888666225224423331

6.539751851761766e+22

As constantes das cadeias de caracteres são representadas em Python delimitadas por aspas. Por exemplo 'bom dia' é uma cadeia de caracteres com 7 caracteres, b, o, m, a, d, i, a.

2.1.2

Expressões

Expressões compostas

Para além das constantes, em Python existe também um certo número de operações, as operações embutidas. Por operações embutidas 5, também conhecidas por operações pré-definidas ou por operações primitivas, entendem-se operações que o Python conhece, independentemente de qualquer indicação que lhe seja dada por um programa. Em Python, para qualquer uma destas operações, existe uma indicação interna (um algoritmo) daquilo que o Python deve fazer quando surge uma expressão com essa operação.

As operações embutidas podem ser utilizadas através do conceito de expressão composta. Informalmente, uma expressão composta corresponde ao conceito de aplicação de uma operação a operandos. Uma expressão composta é constituída por um operador e por um certo número de operandos. Os operadores podem ser unários (se apenas têm um operando, por exemplo, o operador lógico not ou o operador - representando o simétrico) ou binários (se têm dois operandos, por exemplo, + ou *).

Em Python, uma expressão composta é definida sintacticamente do seguinte modo6:

expressão composta ::= operador expressão

operador (expressão)

expressão operador expressão

(expressão operador expressão)

As duas primeiras linhas correspondem à utilização de operadores unários e as 3. Uma cadeia de caracteres é frequentemente designada pelo seu nome em inglês, "string".

4Este carácter corresponde ao espaço em branco.

5Do inglês, "builtin" operations.

6Iremos ver outros modos de definir expressões compostas.

36

CAPÍTULO 2. ELEMENTOS BÁSICOS DE PROGRAMAC

~
,

AO

Prioridade

Operador

Máxima

Aplicação de funções

not, - (simétrico)

*, /, %, and

+, -, or

Mínima

<, >, ==, >=, <=, !=

Tabela 2.1: Prioridade dos operadores em Python.

duas últimas à utilização de operadores binários.

Entre o operador e os operandos podemos inserir espaços em branco para aumentar a legibilidade da expressão, os quais são ignorados pelo Python.

Para os efeitos da apresentação nesta secção, consideramos que um operador corresponde a uma operação embutida.

Utilizando expressões compostas com operações embutidas, podemos originar a seguinte interacção (a qual utiliza operadores cujo significado é óbvio, exceptuando o operador * que representa a multiplicação):

> > > 2012 - 1958

54

```
> > > 3 * (24 + 12)
```

```
108
```

```
> > > 3.0 * (24 + 12)
```

```
108.0
```

```
> > > 7 > 12
```

```
False
```

```
> > > 23 / 7 * 5 + 12.5
```

```
28.928571428571427
```

Uma questão que surge imediatamente quando consideramos expressões compostas diz respeito à ordem pela qual as operações são efectuadas. Por exemplo, qual o denominador da última expressão apresentada? 7 ? $7*5$? $7*5 + 12.5$? É

evidente que o valor da expressão será diferente para cada um destes casos.

Para evitar ambiguidade em relação à ordem de aplicação dos operadores numa expressão, o Python utiliza duas regras que especificam a ordem de aplicação dos operadores. A primeira regra, associada a uma lista de prioridades de ope-

2.1. EXPRESS ~

OES

37

expressão

expressão composta

expressão

operador

expressão

expressão composta

constante

expressão

operador

expressão

expressão composta

constante

expressão

operador

expressão

constante

constante

23

/

7

*

5

+

12.5

Figura 2.1: Estrutura da expressão $23 / 7 * 5 + 12.5$.

radores, especifica que os operadores com maior prioridade são aplicados antes dos operadores com menor prioridade; a segunda regra especifica qual a ordem de aplicação dos operadores quando se encontram dois operadores com a mesma prioridade. Na Tabela 2.1 apresentamos a lista de prioridades dos operadores em Python (estas prioridades são, de modo geral, adoptadas em todas as linguagens de programação). Quando existem dois (ou mais) operadores com a mesma prioridade, eles são aplicados da esquerda para a direita. A utilização de parêntesis permite alterar a ordem de aplicação dos operadores.

Na Figura 2.1 apresentamos a estrutura da expressão $23 / 7 * 5 + 12.5$,

38

CAPÍTULO 2. ELEMENTOS BÁSICOS DE PROGRAMAC

AO

tendo em atenção as regras de aplicação de operadores utilizadas pelo Python.

2.2

Tipos elementares de informação

ao

Em Matemática, é comum classificar as grandezas de acordo com certas características importantes. Existe uma distinção clara entre grandezas reais, grandezas complexas e grandezas do tipo lógico, entre grandezas representando valores individuais e grandezas representando conjuntos de valores, *etc.* De modo análogo, em programação, cada entidade computacional correspondente a um valor pertence a um certo tipo. Este tipo vai caracterizar a possível gama de valores da entidade computacional e as operações a que pode ser sujeita.

A utilização de tipos para caracterizar entidades que correspondem a dados é muito importante em programação. Um tipo de informação é caracterizado por um conjunto de entidades (valores) e um conjunto de operações aplicáveis a essas entidades. Ao conjunto de entidades dá-se nome de domínio do tipo. Cada uma das entidades do domínio do tipo é designada por elemento do tipo.

Os tipos de informação disponíveis variam de linguagem de programação para linguagem de programação. De um modo geral, podemos dizer que os tipos de informação se podem dividir em dois grandes grupos: os tipos elementares e os tipos estruturados. Os tipos elementares são caracterizados pelo facto de as suas constantes (os elementos do tipo) serem tomadas como indecomponíveis (ao nível da utilização do tipo). Como exemplo de um tipo elementar podemos mencionar o tipo lógico, que possui duas constantes, “verdadeiro” e “falso”.

Em contraste, os tipos estruturados s~ao caracterizados pelo facto de as suas constantes serem constituídas por um agregado de valores.

Em Python, como tipos elementares, existem, entre outros, o tipo inteiro, o tipo real e o tipo lógico.

2.2.1

O tipo inteiro

Os números inteiros, em Python designados por `int`, s~ao números sem parte decimal, podendo ser positivos, negativos ou zero. Sobre expressões de tipo inteiro podemos realizar as operações apresentadas na Tabela 2.2. Por exemplo, 7

2.2. TIPOS ELEMENTARES DE INFORMAC

, ~

AO

39

Operação

Tipo dos

Valor

argumentos

$e1 + e2$

Inteiros

A soma dos valores de $e1$ e $e2$.

e1 - e2

Inteiros

O resultado de subtrair e2 a e1.

- e

Inteiro

O simétrico de e.

e1 * e2

Inteiros

O produto dos valores de e1 por e2.

e1 // e2

Inteiros

A divis~ao inteira de e1 por e2.

e1 % e2

Inteiros

O resto da divis~ao inteira de e1 por e2.

abs(e)

Inteiro

O valor absoluto de e.

Tabela 2.2: Opera~oes sobre inteiros em Python.

```
> > > -12
```

```
-12
```

```
> > > 7 // 2
```

```
3
```

```
> > > 7 % 2
```

```
1
```

```
> > > 5 * (7 // 2)
```

```
15
```

```
> > > abs(-3)
```

```
3
```

2.2.2

O tipo real

Os números reais, em Python designados por float8, são números com parte decimal. Em Python, e na maioria das linguagens de programação, existem dois métodos para a representação das constantes do tipo real, a notação decimal e a notação científica.

1. A notação decimal, em que se representa o número, com ou sem sinal, por uma parte inteira, um ponto (correspondente à vírgula), e uma parte decimal. São exemplos de números reais em notação decimal, -7.236, 7.0

e 0.76752. Se a parte decimal ou a parte inteira forem zero, estas podem ser omitidas, no entanto a parte decimal e a parte inteira não podem ser omitidas simultaneamente. Assim, 7. e .1 correspondem a números reais 8Designação que está associada à representação de números reais dentro de um computador, a representação

ao em virgula flutuante (“floating point representation” em inglês).

```
40
```

CAPÍTULO 2. ELEMENTOS BÁSICOS DE PROGRAMAC

~
,

AO

Operação

Tipo dos

Valor

argumentos

$e1 + e2$

Reais

A soma dos valores de $e1$ e $e2$.

e1 - e2

Reais

O resultado de subtrair e2 a e1.

-e

Real

O simétrico de e.

$e1 * e2$

Reais

O produto dos valores de e1 por e2.

$e1 / e2$

Reais

A divisão de e1 por e2.

$\text{abs}(e)$

Real

O valor absoluto de e.

Tabela 2.3: Operações sobre reais em Python.

em Python, respectivamente 7.0 e 0.1.

2. A notação científica em que se representa o número, com ou sem sinal, através de uma mantissa (que pode ser inteira ou real) e de uma potência inteira de dez (o expoente) que multiplicada pela mantissa produz o número. A mantissa e o expoente são separados pelo símbolo “e”.

São exemplos de números reais utilizando a notação científica, 4.2e5

(= 420000.0), -6e-8 (= -0.00000006). A notação científica é utilizada principalmente para representar números muito grandes ou muito pequenos.

Matemática entre o conjunto dos inteiros e o conjunto dos reais, $\mathbb{Z} \subset \mathbb{R}$, não existem deste modo claro em relação à representação de números, num computador os inteiros não estão contidos nos reais. Estes tipos formam conjuntos disjuntos no que respeita à representação das suas constantes. No entanto, as operações definidas sobre números sabem lidar com estas diferentes representações, originando os resultados que seriam de esperar em termos de operações aritméticas.

O que na realidade se passa dentro do computador é que cada operação sobre números (por exemplo, a operação de adição, $+$) corresponde a duas operações internas, uma para cada um dos tipos numéricos (por exemplo, a operação

$+$ $\mathbb{Z}$ que adiciona números inteiros produzindo um número inteiro e a operação

$+$ $\mathbb{R}$ que adiciona números reais produzindo um número real). Estas operações estão associadas à mesma representação externa ($+$). Quando isto acontece, ou seja, quando a mesma representação externa de uma operação está associada a mais do que uma operação dentro do computador, diz-se que a operação está sobrecarregada ⁹.

Quando o Python tem de aplicar uma operação sobrecarregada, determina o tipo de cada um dos operandos. Se ambos forem inteiros, aplica a operação

$+$ $\mathbb{Z}$, se ambos forem reais, aplica a operação $+$ $\mathbb{R}$, se um for inteiro e o outro real, converte o número inteiro para o real correspondente e aplica a operação

$+$ $\mathbb{R}$. Esta conversão tem o nome de coerção¹⁰, sendo demonstrada na seguinte interação:

```
>>> 2 + 3.5
```

```
5.5
```

```
>>> 7.8 * 10
```

⁹Do inglês “overloaded”.

¹⁰Do inglês, “coercion”.

~
,

AO

Operaç~ao

Tipo do

Tipo do

Operaç~ao

argumento

valor

round(e)

Real

Inteiro

O inteiro mais próximo do real e.

int(e)

Real

Inteiro

A parte inteira do real e.

float(e)

Inteiro

Real

O número real correspondente a e.

Tabela 2.4: Transformaç~oes entre reais e inteiros.

78.0

> > > 1 / 3

0.3333333333333333

Note-se que na última expressão, fornecemos dois inteiros à operação / que é definida sobre números reais, pelo que o Python converte ambos os inteiros para reais antes de aplicar a operação, sendo o resultado um número real.

O Python fornece também operações embutidas que transformam números reais em inteiros e vice-versa. Algumas destas operações são apresentadas na Tabela 2.4. A seguinte interacção mostra a utilização destas operações:

```
> > > round(3.3)
```

3

```
> > > round(3.6)
```

4

```
> > > int(3.9)
```

3

```
> > > float(3)
```

3.0

2.2.3

O tipo lógico

O tipo lógico, em Python designado por bool¹¹, apenas pode assumir dois valores, True (verdadeiro) e False (falso).

As operações que se podem efectuar sobre valores lógicos, produzindo valores lógicos, são de dois tipos, as operações unárias e as operações binárias.

¹¹Em honra ao matemático inglês George Boole (1815–1864).

2.3. NOMES E ATRIBUIÇÃO

~
,

AO

43

e1

e2

e1 and e2

e1 or e2

True

True

True

True

True

False

False

True

False

True

False

True

False

False

False

False

Tabela 2.5: Operações de conjunção e disjunção.

- As operações unárias produzem um valor lógico a partir de um valor lógico.

Existe uma operação unária em Python, `not`. A operação `not` muda o valor lógico, de um modo semelhante ao papel desempenhado pela palavra “não”

em português. Assim, `not(True)` tem o valor `False` e `not(False)` tem o valor `True`.

- As operações binárias aceitam dois argumentos do tipo lógico e produzem um valor do tipo lógico. Entre estas operações encontram-se as operações lógicas tradicionais correspondentes à conjunção e à disjunção. A conjunção, representada por `and`, tem o valor `True` apenas se ambos os seus argumentos têm o valor `True` (Tabela 2.5). A disjunção, representada por `or`, tem o valor `False` apenas se ambos os seus argumentos têm o valor `False` (Tabela 2.5).

2.3

Nomes e atribuição

ao

Um dos aspectos importantes em programação corresponde à possibilidade de usar nomes para designar entidades computacionais. A utilização de nomes corresponde a um nível de abstracção no qual deixamos de nos preocupar com a indicação directa da entidade computacional, referindo-nos a essa entidade pelo seu nome. A associação entre um nome e um valor é realizada através da instrução de atribuição, a qual tem uma importância fundamental numa classe de linguagens de programação chamadas linguagens imperativas (de que é exemplo o Python).

A instrução de atribuição em Python recorre à operação embutida `=`, o operador de atribuição. Este operador recebe dois operandos, o primeiro corresponde ao nome que queremos usar para nomear o valor resultante da avaliação do segundo operando, o qual é uma expressão. Em notação BNF, a instrução de atribuição

44

CAPÍTULO 2. ELEMENTOS BÁSICOS DE PROGRAMAC

~
s

AO

é definida do seguinte modo:

❖instrução de atribuição❖ ::= ❖nome❖ = ❖expressão❖

❖nome❖, ❖instrução de atribuição❖, ❖expressão❖

Antes de apresentar a semântica da instrução de atribuição e exemplos da sua utilização, teremos de especificar o que é um nome.

Os nomes s~ao utilizados para representar entidades usadas pelos programas.

Como estas entidades podem variar durante a execu~ao do programa, os nomes s~ao tambem conhecidos por variaveis. Um nome e definido formalmente atraves das seguintes express~oes em notac~ao BNF:

◆nome◆ ::= ◆nome simples◆

◆nome indexado◆

◆nome composto◆

Neste capitulo, apenas consideramos ◆nome simples◆, sendo
◆nome indexado◆

introduzido na Sec~ao 4.1 e ◆nome composto◆ introduzido na
Sec~ao 3.6.

Em Python, um ◆nome simples◆ e uma sequencia de caracteres que
comeca por uma letra ou pelo caracter :

◆nome simples◆ ::= ◆inicial◆ ◆subsequente◆*

◆inicial◆ ::= A B C D E F G H I J K L M N O P Q R

S

T

U

V

X

Y

W

Z

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

x

y

w

z

◆subsequente◆ ::= A B C D E F G H I J K L M N O P

Q

R

S

T

U

V

X

Y

W

Z

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

x

y

w

z

1

2

3

4

5

6

7

8

9

0

Estas expressões em notação BNF dizem-nos que um $\diamond$ nome simples $\diamond$ pode ter tantos caracteres quantos queiramos, tendo necessariamente de começar por uma letra ou pelo carácter . São exemplos de nomes Taxa de juros, Numero, def, factorial, 757. Não

são exemplos de nomes simples 5A (começa por um dígito), turma 10101 (tem um carácter, “ ”, que não é permitido) e igual?

(tem um carácter, “?”, que não é permitido). Para o Python os nomes xpto, Xpto e XPTO são nomes diferentes. Alguns nomes são usados pelo Python, estando reservados pela linguagem para seu próprio uso. Estes nomes, chamados nomes reservados mostram-se na Tabela 2.6.

2.3. NOMES E ATRIBUIÇÃO

~
s

AO

45

and

def

finally

in

or

while

as

del

for

is

pass

with

assert

elif

from

lambda

raise
yield
break
else
global
None
return
class
except
if
nonlocal
True
continue
False
import
not
try

Tabela 2.6: Nomes reservados em Python.

A instrução de atribuição apresenta duas formas distintas. Começamos por discutir a primeira forma, a qual corresponde à primeira linha da expressão BNF que define $\diamond$ instrução de atribuição $\diamond$, a que chamamos atribuição simples.

Ao encontrar a instrução $\diamond$ nome $\diamond = \diamond$ expressão $\diamond$, o Python começa por avaliar a

$\diamond$ expressão $\diamond$ após o que associa $\diamond$ nome $\diamond$ ao valor da $\diamond$ expressão $\diamond$. A execução de uma instrução de atribuição não devolve nenhum valor, mas sim altera o valor de um nome.

A partir do momento em que associamos um nome a um valor (ou nomeamos o valor), o Python passa a “conhecer” esse nome, mantendo uma memória do nome e do valor que lhe está associado. Esta memória correspondente à associação de nomes a valores (ou, de um modo mais geral, entidades computacionais

– de que os valores são um caso particular) tem o nome de ambiente. O ambiente que está associado às operações efectuadas directamente após o carácter de prompt tem o nome de ambiente global. Ao executar uma instrução de atribuição, se o nome não existir no ambiente, o Python insere o nome no ambiente, associando-o ao respectivo valor; se o nome já existir no ambiente, o Python substitui o seu valor pelo valor da expressão.

Consideremos a seguinte interacção com o Python:

```
>>> nota = 17
```

```
>>> nota
```

```
17
```

Na primeira linha surge uma instrução de atribuição. Ao executar esta instrução, o Python avalia a expressão 17 (uma constante) e atribui o seu valor à variável nota. A partir deste momento, o Python passa a “conhecer” o nome, nota, o qual tem o valor 17. A segunda linha da interacção anterior corresponde à avaliação de uma expressão e mostra que se fornecermos ao Python a

```
46
```

CAPÍTULO 2. ELEMENTOS BÁSICOS DE PROGRAMAÇÃO

~
,

AO

Ambiente global

nota

17

Figura 2.2: Representação de um ambiente.

expressão nota (correspondente a um nome), este diz que o seu valor é 17. Este resultado resulta de uma regra de avaliação que afirma que

o valor de um nome é a entidade associada com o nome no ambiente em questão. Na Figura 2.2

mostramos a representação do ambiente correspondente a esta interação. Um ambiente é representado por um retângulo a cinzento, dentro do qual aparecem associações de nomes a entidades computacionais. Cada associação contém um nome, apresentado no lado esquerdo, ligado por uma seta ao seu valor.

Consideremos agora a seguinte interação com o Python, efectuada depois da interação anterior:

```
> > > nota = nota + 1
```

```
> > > nota
```

```
18
```

```
> > > soma
```

```
NameError: name 'soma' is not defined
```

Segundo a semântica da instrução de atribuição, para a instrução apresentada na primeira linha, o Python começa por avaliar a expressão `nota + 1`, a qual tem o valor 18, em seguida associa o nome `nota` a este valor, resultando no ambiente apresentado na Figura 2.3. A instrução de atribuição `nota = nota`

+ 1 tem o efeito de atribuir à variável `nota` o valor anterior de `nota` mais um.

Este último exemplo mostra o carácter dinâmico da operação de atribuição: em primeiro lugar, a expressão à direita do símbolo `=` é avaliada, e, em segundo lugar, o valor resultante é atribuído à variável à esquerda deste símbolo. Isto mostra que uma operação de atribuição não é uma equação matemática, mas sim um processo de atribuir o valor da expressão à direita do operador de atribuição à variável à sua esquerda. Na interação anterior, mostramos também que se fornecermos ao Python um nome que não existe no ambiente (`soma`), o Python gera um erro, dizendo que não conhece o nome.

Todos os nomes que são definidos pela operação de atribuição, directamente

2.3. NOMES E ATRIBUIC

AO

47

Ambiente global

nota

18

Figura 2.3: Ambiente resultante da execuç~ao de $\text{nota} = \text{nota} + 1$.

Ambiente global

pi

3.14

raio

5

area

78.5

Figura 2.4: Ambiente depois da definiç~ao de pi, raio e area.

executada após o carácter de pronto, pertencem a um ambiente a que se chama ambiente global. O ambiente global (também conhecido por espaço de nomes¹²) contém associaç~oes para todos os nomes que foram fornecidos directamente ao Python. Para além disso, o ambiente global contém também os nomes de todas as operaç~oes embutidas do Python, ou seja, as operaç~oes que fazem parte do Python. O ambiente global é criado quando uma sess~ao com o Python é iniciada e existe enquanto essa sess~ao durar.

Na seguinte interacç~ao com o Python, começamos por tentar atribuir o valor 3 ao nome `def`, o qual corresponde a um nome reservado do Python (ver Tabela 2.6).

O Python reage com um erro. Seguidamente, definimos valores para as variáveis `pi`, `raio` e `area`, resultando no ambiente apresentado na Figura 2.4.

```
> > > def = 3
```

Syntax Error: def = 3: <string>, line 15

```
> > > pi = 3.14
```

```
> > > raio = 5
```

```
> > > area = pi raio raio
```

```
> > > raio
```

5

```
> > > area
```

12Do inglês “name space”.

48

CAPÍTULO 2. ELEMENTOS B ÁSICOS DE PROGRAMAC

~
,

AO

Ambiente global

pi

3.14

raio

10

area

78.5

Figura 2.5: Ambiente depois da alteraç~ao do valor de raio.

78.5

```
> > > raio = 10
```

```
> > > area
```

A interacção anterior também mostra que se mudarmos o valor da variável raio o valor de area, embora tenha sido calculado a partir do nome raio, não se altera (Figura 2.5).

Consideremos agora a segunda forma da instrução de atribuição, a qual é conhecida por atribuição múltipla e que corresponde à segunda linha da expressão BNF que define *instrução de atribuição*. Ao encontrar uma instrução da forma

nome1, *nome2*, ..., *nomen* = *exp1*, *exp2*, ..., *expn*, o Python começa por avaliar as expressões *exp1*, *exp2*, ... *expn* (a ordem da avaliação destas expressões é irrelevante), após o que associa *nome1* ao valor da expressão *exp1*, *nome2* ao valor da expressão *exp2*, ... *nomen* ao valor da expressão *expn*.

O funcionamento da instrução de atribuição múltipla é ilustrado na seguinte interacção:

```
> > > nota_teste1, nota_teste2, nota_projecto = 15, 17, 14
```

```
> > > nota_teste1
```

```
15
```

```
> > > nota_teste2
```

```
17
```

```
> > > nota_projecto
```

2.3. NOMES E ATRIBUIC

~
s

AO

49

Ambiente global

a

2

b

17.5

Figura 2.6: Ambiente global.

14

Consideremos agora a seguinte interação

```
>>> nota_1, nota_2 = 17, nota_1 + 1
```

```
NameError: name 'nota_1' is not defined
```

e analisemos a origem do erro. Dissemos que o Python começa por avaliar as expressões à direita do símbolo `=`, as quais são 17 e `nota_1 + 1`. O valor da constante 17 é 17. Ao avaliar a expressão `nota_1 + 1`, o Python não encontra o valor de `nota_1` no ambiente, pelo que gera um erro semelhante ao apresentado na página 46.

Consideremos agora a seguinte interação:

```
>>> a = 2
```

```
>>> b = 17.5
```

```
>>> a
```

2

```
>>> b
```

17.5

```
>>> a, b = b, a
```

```
>>> a
```

17.5

```
>>> b
```

2

Ao executar a instrução `a, b = b, a`, o Python começa por avaliar as expressões `b` e `a`, cujos valores são, respectivamente 17.5 e 2 (Figura 2.6). Em seguida, o valor 17.5 é atribuído à variável `a` e o valor 2 é atribuído à variável `b`

CAPÍTULO 2. ELEMENTOS BÁSICOS DE PROGRAMAC

AO

Ambiente global

a

17.5

b

2

Figura 2.7: Ambiente global depois da execução de `a, b = b, a`.

(Figura 2.7). Ou seja a instrução `a, b = b, a` tem o efeito de trocar os valores das variáveis `a` e `b`.

A instrução de atribuição é a primeira instrução do Python que considerámos.

Ao passo que uma expressão tem um valor, e consequentemente quando nos referimos às acções realizadas pelo Python para calcular o valor de uma expressão dizemos que a expressão é avaliada, uma instrução não tem um valor mas causa a realização de certas acções, por exemplo a atribuição de um nome a uma variável. Por esta razão, quando nos referimos às acções efectuadas pelo Python associadas a uma instrução dizemos que a instrução é executada.

2.4

Predicados e condições

oes

Uma operação que produz resultados do tipo lógico chama-se um predicado.

Uma expressão cujo valor é do tipo lógico chama-se uma condição. As condições podem ser combinadas através de operações lógicas. Entre outros, no Python existem, como operações embutidas, os operadores relacionais que se apresentam na Tabela 2.7. As operações

lógicas foram apresentadas na Secção 2.2.3.

A seguinte interacção com o Python mostra a utilização de operadores relacionais e de operações lógicas (para compreender a última expressão fornecida ao Python, recorde-se a prioridade das operações apresentada na Tabela 2.1.):

```
> > > nota = 17
```

```
> > > 3 < nota % 2
```

False

```
> > > 3 < nota // 2
```

True

```
> > > 4 > 5 or 2 < 3
```

2.4. PREDICADOS E CONDIC

~
,

OES

51

Operação

Tipo dos

Valor

argumentos

$e1 == e2$

Números

Tem o valor True se e só se os valores das expressões $e1$ e $e2$ são iguais.

$e1 != e2$

Números

Tem o valor True se e só se os valores das

expressões $e1$ e $e2$ são diferentes.

$e1 > e2$

Números

Tem o valor True se e só se o valor da expressão $e1$ é maior do que o valor da expressão $e2$.

$e1 \geq e2$

Números

Tem o valor True se e só se o valor da expressão $e1$ é maior ou igual ao valor da expressão $e2$.

$e1 < e2$

Números

Tem o valor True se e só se o valor da expressão $e1$ é menor do que o valor da expressão $e2$.

$e1 \leq e2$

Números

Tem o valor True se e só se o valor da expressão $e1$ é menor ou igual ao valor da expressão $e2$.

Tabela 2.7: Operadores relacionais em Python.

True

O Python permite simplificar algumas operações envolvendo operadores relacionais. Consideremos a expressão $1 < 3 < 5$ que normalmente é usada em Matemática. Considerando a sua tradução directa para Python, $1 < 3 < 5$, e a prioridade dos operadores, esta expressão deverá ser avaliada da esquerda para a direita $(1 < 3) < 5$, dando origem a $\text{True} < 5$ que corresponde a uma expressão que não faz sentido avaliar. Na maioria das linguagens de programação, esta expressão deverá ser traduzida para $(1 < 3) \text{ and } (3 < 5)$. O Python oferece-nos uma notação simplificada para escrever condições com operadores relacionais, assim a expressão $(1 < 3) \text{ and } (3 < 5)$ pode ser simplificada para $1 < 3 < 5$, como o mostram os seguintes exemplos:

> > > 2 < 4 < 6 < 9

True

> > > 2 < 4 > 3 > 1 < 12

True

Uma alternativa para tornar a notação mais simples, como a que acabámos de apresentar relativa aos operadores relacionais, é vulgarmente designada por açúcar sintáctico¹³.

¹³Do inglês, “syntactic sugar”.

52

CAPÍTULO 2. ELEMENTOS BÁSICOS DE PROGRAMAÇÃO

~
,

AO

2.5

Comunicação

ao com o exterior

2.5.1

Leitura de dados

Durante a execução de um programa, o computador necessita normalmente de obter valores do exterior para efectuar a manipulação da informação. A obtenção de valores do exterior é feita através das operações de leitura de dados.

As operações de leitura de dados permitem transmitir informação do exterior para o programa. Por “exterior” entenda-se (1) o mundo exterior ao programa, por exemplo, um ser humano, ou (2) o próprio computador, por exemplo, um ficheiro localizado dentro do computador. Neste capítulo apenas consideramos operações de leitura de dados em que os dados são fornecidos através do teclado. No Capítulo 9, consideramos operações de leitura de dados localizados em ficheiros.

O Python fornece uma operação de leitura de dados, a função input.

Esta função tem a seguinte sintaxe:

```
◆ leitura de dados ◆ ::= input()
```

```
input(◆informaç~ao◆)
```

```
◆informaç~ao◆ ::= ◆cadeia de caracteres◆
```

Ao encontrar a função `input(◆informaç~ao◆)` o Python mostra no écran a cadeia de caracteres correspondente a `◆informaç~ao◆`, após o que lê todos os símbolos introduzidos através do teclado até que o utilizador carregue na tecla “Return”.

A sequência de caracteres encontrada durante a leitura, é devolvida pela função `input` como uma cadeia de caracteres.

Consideremos as seguintes interacções:

```
> > > input('-> ')
```

```
-> 5
```

```
'5'
```

Neste caso o Python mostra o valor da cadeia de caracteres `'-> '`, o qual corresponde a `->` , e lê o que é fornecido através do teclado, neste caso, `5`

seguido de “Return”, sendo devolvida a cadeia de caracteres `'5'`.

```
> > > input()
```

2.5. COMUNICAC

~
,

AO COM O EXTERIOR

53

Carácter escape Significado

\\

Barra ao contrário (\)

\'

Plica (')

\"

Aspas (")

\a

Toque de campainha

\b

Retrocesso de um espaço

\f

Salto de página

\n

Salto de linha

\r

“Return”

\t

Tabulaç~ao horizontal

\v

Tabulaç~ao vertical

Tabela 2.8: Alguns caracteres de escape em Python.

estou a escrever sem carácter de pronto

'estou a escrever sem carácter de pronto'

Neste caso não é mostrada nenhuma indicaç~ao no écran do computador, ficando o Python à espera que seja escrita qualquer informaç~ao. Escrevendo no teclado “estou a escrever sem carácter de pronto” seguido de “Return”, a funç~ao input devolve a cadeia de caracteres 'estou a escrever sem carácter de pronto', e daí a aparente duplicaç~ao das duas últimas linhas na interacç~ao anterior.

> > > input('Por favor escreva qualquer coisa\n-> ') Por favor

escreva qualquer coisa

```
-> 554 umas palavras 3.14
```

```
'554 umas palavras 3.14'
```

O exemplo anterior, introduz algo de novo. Na cadeia de caracteres que é fornecida à função `input` aparece algo que não vimos até agora, a sequência

`\n`. A isto chama-se um carácter de escape¹⁴. Um carácter de escape é um carácter não gráfico com um significado especial para um meio de escrita, por exemplo, uma impressora ou o ecrã do computador. Em Python, um carácter de escape corresponde a um carácter precedido por uma barra ao contrário, “`\`”.

Na Tabela 2.8 apresentam-se alguns caracteres de escape existentes em Python.

¹⁴Do inglês “escape character”.

54

CAPÍTULO 2. ELEMENTOS BÁSICOS DE PROGRAMAC

~
,

AO

Finalmente, a seguinte interação mostra a atribuição do valor lido a uma variável, `a`.

```
>>> a = input('-> ')
```

```
-> teste
```

```
>>> a
```

```
'teste'
```

Sendo o valor da função `input` uma cadeia de caracteres, poderemos questionar como ler valores inteiros ou reais. Para isso teremos que recorrer à função embutida, `eval`, chamada a função de avaliação, a qual tem a seguinte sintaxe:

◆função de avaliação◆ ::= `eval(◆cadeia de caracteres◆)` A função `eval` recebe uma cadeia de caracteres e devolve o resultado de

avaliar essa cadeia de caracteres como sendo um expressão. Por exemplo:

```
> > > eval('3 + 5')
```

8

```
> > > eval('2')
```

2

```
> > > eval('5 * 3.2 + 10')
```

26.0

```
> > > eval('fundamentos da programaç~
```

ao')

Syntax Error: fundamentos da programaç~

ao: <string>, line 114

Combinando a função de avaliação com a função input podemos obter o seguinte resultado:

```
> > > b = eval(input('Escreva um número: ')) Escreva um número:
```

4.56

```
> > > b
```

4.56

2.5.2

Escrita de dados

Após efectuar a manipulação da informação, é importante que o computador possa comunicar ao exterior os resultados a que chegou. Isto é feito através

2.5. COMUNICAC

~
,

AO COM O EXTERIOR

55

das operações de escrita de dados. As operações de escrita de dados permitem transmitir informação do programa para o exterior. Por “exterior” entenda-se (1) o mundo exterior ao programa, por exemplo, um ser humano, ou (2) o próprio computador, por exemplo, um ficheiro localizado dentro do computador.

Neste capítulo apenas consideramos operações de escrita de dados em que os dados são escritos no écran. No Capítulo 9, consideramos operações de escrita de dados para ficheiros.

Em Python existe a função embutida, `print`, com a sintaxe definida pelas seguintes expressões em notação BNF:

◆escrita de dados◆ ::= `print()`

`print(◆expressões◆)`

◆expressões◆ ::= ◆expressão◆

◆expressão◆, ◆expressões◆

Ao encontrar a função `print()`, o Python escreve uma linha em branco no écran. Ao encontrar a função `print(◆exp1◆, ... ◆expn◆)`, o Python começa por avaliar cada uma das expressões `◆exp1◆ ... ◆expn◆`, após o que as escreve na mesma linha do écran, separadas por um espaço em branco.

A seguinte interação ilustra o uso da função `print`:

```
>>> a = 10
```

```
>>> b = 15
```

```
>>> print('a = ', a, 'b = ', b)
```

```
a =
```

```
10 b =
```

```
15
```

```
>>> print('a = ', a, '\nb = ', b)
```

```
a =
```

```
10
```

b =

15

Sendo print uma função, estamos à espera que esta devolva um valor. Se imediatamente após a interacção anterior atribuirmos à variável c o valor devolvido por print constatamos o seguinte¹⁵:

```
> > > c = print('a = ', a, '\nb = ', b)
```

```
> > > print(c)
```

None

15No capítulo 13 voltamos a considerar a constante None.

56

CAPÍTULO 2. ELEMENTOS BÁSICOS DE PROGRAMAC

~
,

AO

Ou seja, print é uma função que não devolve nada! Em Python, Existem algumas funções cujo objectivo não é a produção de um valor, mas sim a produção de um efeito, a alteração de qualquer coisa. A função print é uma destas funções.

Depois da avaliação da função print, o conteúdo do ecrã do nosso computador muda, sendo esse efeito a única razão da existência desta função.

2.6

Programas, instruções

es e sequências

ao

Até agora a nossa interacção com o Python correspondeu a fornecer comandos (através de expressões e de uma instrução) ao Python, imediatamente a seguir ao carácter de pronto, e a receber a resposta do Python ao comando fornecido. De um modo geral, para instruímos o Python a realizar uma dada tarefa fornecemos-lhe um programa, uma sequência de comandos, que o Python executa, comando a

comando.

Um programa em Python corresponde a uma sequência de zero ou mais definições seguida por uma sequência de uma ou mais instruções. Em notação BNF, um programa em Python, também conhecido por um guião, é definido do seguinte modo:

$\text{programa em Python} ::= \text{definição}^* \text{instrução}^+$

Um programa não contém directamente expressões, aparecendo estas associadas a definições e a instruções.

O conceito de definição é apresentado nos capítulos 3 e 10, pelo que os nossos programas neste capítulo apenas contêm instruções. Nesta secção, começamos a discutir os mecanismos através dos quais se pode especificar a ordem de execução das instruções de um programa e apresentar algumas das instruções que podemos utilizar num programa em Python. No fim do capítulo, estaremos aptos a desenvolver alguns programas em Python, extremamente simples mas completos.

O controle da sequência de instruções a executar durante um programa joga um papel essencial no funcionamento de um programa. Por esta razão, as linguagens de programação fornecem estruturas que permitem especificar qual a ordem de execução das instruções do programa.

Do inglês “script”.

2.6. PROGRAMAS, INSTRUÇÕES E SEQUÊNCIAS

INSTRUÇÕES E SEQUÊNCIAS

AO

57

Ao nível da linguagem máquina, existem dois tipos de estruturas de controle: a sequência e o salto. A sequência especifica que as instruções de um programa são executadas pela ordem em que aparecem no programa. O salto especifica a transferência da execução para qualquer ponto do programa. As linguagens de alto nível, de que

o Python é um exemplo, para além da sequenciação e do salto (a instrução de salto pode ter efeitos perniciosos na execução de um programa e não será considerada neste livro), fornecem estruturas de controle mais sofisticadas, nomeadamente a selecção e a repetição.

A utilização de estruturas de controle adequadas contribui consideravelmente para a facilidade de leitura e manutenção de programas. De facto, para dominar a compreensão de um programa é crucial que as suas instruções sejam estruturadas de acordo com processos simples, naturais e bem compreendidos.

A sequenciação é a estrutura de controle mais simples, e consiste na especificação de que as instruções de um programa são executadas sequencialmente, pela ordem em que aparecem no programa. Sendo $\diamond inst1 \diamond$ e $\diamond inst2 \diamond$ símbolos não terminais que correspondem a instruções, a indicação de que a instrução $\diamond inst2 \diamond$

é executada imediatamente após a execução da instrução $\diamond inst1 \diamond$ é especificada, em Python escrevendo a instrução $\diamond inst2 \diamond$ numa linha imediatamente a seguir à linha em que aparece a instrução $\diamond inst1 \diamond$. Ou seja, o fim de linha representa implicitamente o operador de sequenciação. O conceito de sequência de instruções é definido através da seguinte expressão em notação BNF:

$$\diamond instruções \diamond := \diamond instrução \diamond CR$$
$$\diamond instrução \diamond CR \diamond instruções \diamond$$

Nesta definição, CR é um símbolo terminal que corresponde ao símbolo obtido carregando na tecla “Return” do teclado, ou seja, CR corresponde ao fim de uma linha,

Consideremos o seguinte programa (sequência de instruções):

```
nota_1 = 17
```

```
nota_2 = 18
```

```
media = (nota_1 + nota_2) / 2
```

```
print('A média é ', media)
```

Se instruímos o Python para executar este programa, este escreve

58

CAPÍTULO 2. ELEMENTOS BÁSICOS DE PROGRAMAC

AO

A média é

17.5

o que mostra um exemplo da execução sequencial de instruções.

2.7

Selecc~

ao

Para desenvolver programas complexos, é importante que possamos especificar a execução condicional de instruções: devemos ter a capacidade de decidir se uma instrução ou grupo de instruções deve ou não ser executado, dependendo do valor de uma condição. Esta capacidade é por nós utilizada constantemente.

Repare-se, por exemplo, nas instruções para o papagaio voador, apresentadas no Capítulo 1, página 8: “se o vento soprar forte deverá prender na argola inferior.

Se o vento soprar fraco deverá prender na argola superior.”.

A instrução if17 permite a selecção entre duas ou mais alternativas. Dependendo do valor de uma condição, esta instrução permite-nos seleccionar uma de duas ou mais instruções para serem executadas. A sintaxe da instrução if é definida pelas seguintes expressões em notação BNF18:

◆instrução “if”◆ ::= if ◆condição◆: CR

◆instrução composta◆

◆outras alternativas◆*

{◆alternativa final◆}

◆outras alternativas◆ := elif ◆condição◆: CR

◆instrução composta◆

◆alternativa final◆ := else: CR

◆instruç~ao composta◆

◆instruç~ao composta◆ := TAB + ◆instruç~oes◆ TAB-

◆condiç~ao◆ := ◆express~ao◆

Nesta definiç~ao, TAB+ corresponde ao s~imbolo obtido carregando na tecla que efectua a tabulaç~ao e TAB-corresponde ao s~imbolo obtido desfazendo a acç~ao correspondente a TAB+ . Numa instruç~ao composta, o efeito de TAB+ aplica-se a cada uma das suas instruç~oes, fazendo com que estas comecem todas na mesma 17A palavra “if” traduz-se, em português, por “se”.

18Existem outras alternativas para a instruç~ao if que n~ao consideramos neste livro.

2.7. SELECC

~
,

AO

59

coluna. Este aspecto é conhecido por paragrafaç~ao. Uma instruç~ao if estende-se por várias linhas, n~ao deixando de ser considerada uma única instruç~ao.

Na definiç~ao sintáctica da instruç~ao if, a ◆condiç~ao◆ representa qualquer express~ao do tipo lógico. Este último aspecto, a imposiç~ao de que express~ao seja do tipo lógico, n~ao pode ser feita recorrendo a express~oes em notaç~ao BNF.

De acordo com esta definiç~ao,

if nota > 15:

print('Bom trabalho')

corresponde a instruç~oes if, mas

if nota > 15:

print('Bom trabalho')

n~ao corresponde a uma instruç~ao if, pois falta o s~imbolo terminal TAB+ antes da instruç~ao print('Bom trabalho').

Para apresentar a semântica da instrução `if` vamos considerar cada uma das suas formas:

1. Ao encontrar uma instrução da forma

`if cond :`

`instruções`

o Python começa por avaliar a expressão `cond`. Se o seu valor for `True`, o Python executa as instruções correspondentes a `instruções`; se o valor da expressão `cond` for `False`, o Python não faz mais nada relativamente a esta instrução `if`. Os efeitos desta semântica são mostrados na seguinte interação:

```
>>> if 3 < 5 :
```

```
...
```

```
print('3 < 5')
```

```
...
```

```
3 < 5
```

```
>>> if 3 > 5 :
```

```
...
```

```
print('3 < 5')
```

```
...
```

```
>>>
```

```
60
```

CAPÍTULO 2. ELEMENTOS BÁSICOS DE PROGRAMAÇÃO

~
,

AO

Nesta interação com o Python, surge um novo carácter de pronto, correspondente a `...`. Esta nova forma do carácter de pronto corresponde à indicação do Python que está à espera de mais informação para terminar de ler o comando que está a ser fornecido.

Esta informação termina quando o utilizador escreve uma linha em branco.

A última linha, apenas com o sinal de pronto, indica que a instrução na linha anterior foi executada, estando o Python à espera de receber novo comando.

2. Ao encontrar uma instrução da forma (a qual não usa outras alternativas, utilizando apenas a alternativa final)

```
if cond :
```

```
    instruções1
```

```
else :
```

```
    instruções2
```

o Python começa por avaliar a expressão `cond`. Se o seu valor for True, as instruções correspondentes a `instruções1` são executadas e as instruções correspondentes a `instruções2` não o são; se o seu valor for False, as instruções correspondentes a `instruções2` são executadas e as instruções correspondentes a `instruções1` não o são.

A seguinte interação mostra a semântica da instrução `if`:

```
>>> nota = 15
```

```
>>> if nota > 16 :
```

```
...
```

```
print('Bom trabalho')
```

```
... else:
```

```
...
```

```
print('Podes fazer melhor')
```

```
...
```

```
print('Estuda mais')
```

```
...
```


Podes fazer melhor

Estuda mais

```
>>> nota = 17
```

```
>>> if nota > 16 :
```

```
...
```

```
print('Bom trabalho')
```

```
... else:
```

```
...
```

```
print('Podes fazer melhor')
```

```
...
```

```
print('Estuda mais')
```

2.7. SELECC

~
,

AO

61

...

Bom trabalho

3. De um modo geral, ao encontrar uma instruç~ao da forma: if

◆cond1◆ :

◆instruç~oes1◆

elif ◆cond2◆ :

◆instruç~oes2◆

elif ◆cond3◆ :

◆instruç~oes3◆

...

else :

◆instruções◆

o Python começa por avaliar a expressão ◆cond1◆. Se o seu valor for True, as instruções correspondentes a ◆instruções1◆ são executadas e a execução da instrução if termina; se o seu valor for False, o Python avalia a expressão ◆cond2◆. Se o seu valor for True, as instruções correspondentes a

◆instruções2◆ são executadas e a execução da instrução if termina. Em caso contrário, o Python avalia a expressão ◆cond3◆, e assim sucessivamente. Se todas as condições forem falsas, o Python executa as instruções correspondentes a ◆instruções◆.

Consideremos a seguinte instrução if, a qual para um valor quantitativo de uma nota escreve o seu valor qualitativo equivalente, usando a convenção de que uma nota entre zero e 4 corresponde a mau, uma nota entre 5 e 9 corresponde a medíocre, uma nota entre 10 e 13 corresponde a suficiente, uma nota entre 14 e 17 corresponde a bom e uma nota superior a 17 corresponde a muito bom. Como exercício, o leitor deve convencer-se que embora o limite inferior das gamas de notas não seja verificado, esta instrução if realiza adequadamente o seu trabalho para qualquer valor de nota.

if nota <= 4 :

print('Mau')

elif nota <= 9 :

print('Mediocre')

elif nota <= 13 :

62

CAPÍTULO 2. ELEMENTOS BÁSICOS DE PROGRAMAÇÃO

~
,

AO

print('Suficiente')

elif nota <= 17 :

```
print('Bom')
```

```
else :
```

```
print('Muito bom')
```

2.8

Repetição

ao

Em programação é frequente ser preciso repetir a execução de um grupo de instruções, ou mesmo repetir a execução de todo o programa, para diferentes valores dos dados. Repare-se, por exemplo, na receita para os rebuçados de ovos, apresentada no Capítulo 1, página 7: “Leva-se o açúcar ao lume com um copo de água e deixa-se ferver até fazer ponto de pérola”, esta instrução corresponde a dizer que “enquanto não se atingir o ponto de pérola, deve-se deixar o açúcar com água a ferver”.

Em programação, uma sequência de instruções executada repetitivamente é chamada um ciclo. Um ciclo é constituído por uma sequência de instruções, o corpo do ciclo, e por uma estrutura que controla a execução dessas instruções, especificando quantas vezes o corpo do ciclo deve ser executado. Os ciclos são muito comuns em programação, sendo raro encontrar um programa sem um ciclo.

Cada vez que as instruções que constituem o corpo do ciclo são executadas, dizemos que se efectuou uma passagem pelo ciclo. As instruções que constituem o corpo do ciclo podem ser executadas qualquer número de vezes (eventualmente, nenhuma) mas esse número tem de ser finito. Há erros semânticos que podem provocar a execução interminável do corpo do ciclo, caso em que se diz que existe um ciclo infinito¹⁹. Em Python, existem duas instruções que permitem a especificação de ciclos, a instrução `while`, que apresentamos nesta secção, e a instrução `for`, que apresentamos no Capítulo 4.

A instrução `while` permite especificar a execução repetitiva de um conjunto de instruções enquanto uma determinada expressão do tipo lógico tiver o valor verdadeiro. A sintaxe da instrução `while` é definida pela seguinte expressão em 190 conceito de ciclo infinito é teórico, pois, na prática ciclo acabará por terminar ou porque nós o interrompemos (fartamo-nos de esperar) ou porque os recursos computacionais se esgotam.

AO

63

notação BNF20:

$$\text{instrução while} ::= \text{while expressão} : \text{CR}$$

$$\text{instrução composta}$$

Na definição sintáctica da instrução while, a expressão representa uma expressão do tipo lógico. Este último aspecto não pode ser explicitado recorrendo a expressões em notação BNF.

Na instrução while $\text{cond} : \text{CR inst}$, o símbolo não terminal inst corresponde ao corpo do ciclo e o símbolo cond , uma expressão do tipo lógico, representa a condição que controla a execução do ciclo.

A semântica da instrução while é a seguinte: ao encontrar a instrução while

$\text{cond} : \text{CR inst}$, o Python calcula o valor de cond . Se o seu valor for True, o Python efectua uma passagem pelo ciclo, executando as instruções correspondentes a inst . Em seguida, volta a calcular o valor de cond e o processo repete-se enquanto o valor de cond for True. Quando o valor de cond for False, a execução do ciclo termina.

É evidente que a instrução que constitui o corpo do ciclo deve modificar o valor da expressão que controla a execução do ciclo, caso contrário, o ciclo pode nunca terminar (se o valor desta expressão é inicialmente True e o corpo do ciclo não modifica esta expressão, então estamos em presença de um ciclo infinito).

No corpo do ciclo, pode ser utilizada uma outra instrução existente em Python, a instrução break. A instrução break apenas pode aparecer dentro do corpo de um ciclo. A sintaxe da instrução break é definida pela seguinte expressão em notação BNF21:

$$\text{instrução break} ::= \text{break}$$

Ao encontrar uma instrução break, o Python termina a execução do

ciclo, independentemente do valor da condição que controla o ciclo.

Como exemplo da utilização de uma instrução while, vulgarmente designada por um ciclo while, consideremos a seguinte sequência de instruções: soma = 0

```
num = eval(input('Escreva um inteiro\n(-1 para terminar): '))
```

 20A palavra “while” traduz-se em português por “enquanto”.

21A palavra “break” traduz-se em português por “fuga”, “quebra”, “pausa”, ou “ruptura”.

64

CAPÍTULO 2. ELEMENTOS BÁSICOS DE PROGRAMAÇÃO

~
,

AO

```
while num != -1 :
```

```
soma = soma + num
```

```
num = eval(input('Escreva um inteiro\n(-1 para terminar): ')) print('A soma é: ', soma)
```

Estas instruções começam por estabelecer o valor inicial da variável soma como sendo zero, após o que efectuam a leitura de uma sequência de números inteiros positivos, calculando a sua soma. A quantidade de números a ler é desconhecida à partida. Para assinalar que a sequência de números terminou, fornecemos ao Python um número especial, no nosso exemplo, o número -1. Este valor é chamado o valor sentinela, porque assinala o fim da sequência a ser lida.

Neste programa inserimos linhas em branco para separar partes do programa, a inicialização de variáveis, o ciclo, e a escrita do resultado. As linhas em branco podem ser inseridas entre instruções, aumentando a facilidade de leitura do programa, mas não tendo qualquer efeito sobre a sua execução. As linhas em branco podem ser consideradas como a instrução vazia, a qual é definida pela seguinte sintaxe:

◆instrução vazia◆ :: =

Ao encontrar uma instrução vazia, o Python não toma nenhuma

acção.

Este programa utiliza uma instrução while, a qual é executada enquanto o número fornecido pelo utilizador for diferente de -1. O corpo deste ciclo é constituído por uma instrução composta, que é, por sua vez, constituída por duas instruções soma = soma + num (que actualiza o valor da soma) e num

= eval(input('Escreva um inteiro\n(-1 para terminar): ')) (que lê o número fornecido). Note-se que este ciclo pressupõe que já foi fornecido um número ao computador, e que o corpo do ciclo inclui uma instrução (num =

eval(input('Escreva um inteiro\n(-1 para terminar): '))) que modifica o valor da condição que controla o ciclo (num != -1).

Existem dois aspectos importantes a lembrar relativamente à instrução while: 1. De um modo geral, o número de vezes que o corpo do ciclo é executado não pode ser calculado antecipadamente: a condição que especifica o término 22Esta operação é conhecida por inicialização da variável.

2.8. REPETIC

AO

65

do ciclo é testada durante a execução do próprio ciclo, sendo impossível saber de antemão como vai prosseguir a avaliação.

2. Pode acontecer que o corpo do ciclo não seja executado nenhuma vez. Com efeito, a semântica da instrução while especifica que o valor da expressão que controla a execução do ciclo é calculado antes do início da execução do ciclo. Se o valor inicial desta expressão é False, o corpo do ciclo não é executado.

Consideremos agora um programa que lê um inteiro positivo e calcula a soma dos seus dígitos. Este programa lê um número inteiro, atribuindo-o à variável num e inicializa o valor da soma dos dígitos (variável soma) para zero.

Após estas duas acções, o programa executa um ciclo enquanto o número não for zero. Neste ciclo, adiciona sucessivamente cada um

dos algarismos do número à variável soma. Para realizar esta tarefa o programa tem que obter cada um dos dígitos do número. Notemos que o dígito das unidades corresponde ao resto da divisão inteira do número por 10 ($\text{num \% 10}$). Após obter o algarismo das unidades, o programa tem que “remover” este algarismo do número, o que pode ser feito através da divisão inteira do número por 10 (num // 10). Depois do ciclo, o programa escreve o valor da variável soma.

O seguinte programa realiza a tarefa desejada: $\text{num} =$
 $\text{eval}(\text{input}(\text{'Escreva um inteiro positivo\n? '}))$ $\text{soma} = 0$

```
while num > 0 :
```

```
    digito = num % 10
```

```
    # obtém o algarismo das unidades
```

```
    num = num // 10
```

```
    # remove o algarismo das unidades
```

```
    soma = soma + digito
```

```
print('Soma dos d'
```

```
ígitos: ', soma)
```

Este programa contém anotações, comentários, que explicam ao seu leitor algumas das instruções do programa. Comentários são frases em linguagem natural que aumentam a facilidade de leitura do programa, explicando o significado das variáveis, os objectivos de zonas do programa, certos aspectos do algoritmo, *etc.*

66

CAPÍTULO 2. ELEMENTOS BÁSICOS DE PROGRAMAC

~
s

AO

Os comentários podem também ser expressões matemáticas provando propriedades sobre o programa. Em Python, um comentário é qualquer linha, ou parte de linha, que se encontra após o símbolo “#”. Tal como as linhas em branco, os comentários são ignorados pelo computador.

Notas finais

Começámos por apresentar alguns tipos existente em Python, os tipos inteiro, real, lógico, e as cadeias de caracteres. Vimos que um tipo corresponde a um conjunto de valores, juntamente com um conjunto de operações aplicáveis a esses valores. Definimos alguns dos componentes de um programa, nomeadamente, as expressões e algumas instruções elementares.

Neste capítulo, apresentámos a estrutura de um programa em Python. Um programa é constituído, opcionalmente por uma sequência de definições (que ainda não foram abordadas), seguido por uma sequência de instruções. Estudámos três instruções em Python, a instrução de atribuição (que permite a atribuição de um valor a um nome), a instrução if (que permite a selecção de grupos de instruções para serem executadas) e a instrução while (que permite a execução repetitiva de um conjunto de instruções). Apresentámos também duas operações para efectuarem a entrada e saída de dados.

2.10

Exercícios

1. Escreva um programa em Python que pede ao utilizador que lhe forneça um inteiro correspondente a um número de segundos e que calcula o número de dias correspondentes a esse número de segundos. O número de dias é fornecido como um real. O programa termina quando o utilizador fornece um número negativo. O seu programa deve possibilitar a seguinte interacção:

Escreva um número de segundos

(um número negativo para terminar)

? 45

O número de dias correspondente é

0.0005208333333333333

Escreva um número de segundos

2.10. EXERCÍCIOS

(um número negativo para terminar)

? 6654441

O número de dias correspondente é

77.01899305555555

Escreva um número de segundos

(um número negativo para terminar)

? -1

> > >

2. Escreva um programa em Python que lê três números e que diz qual o maior dos números lidos.

3. A conversão de temperatura em graus Fahrenheit (F) para graus centígrados (C) é dada através da expressão

5

$$C = 9 \cdot (F - 32).$$

Escreva um programa em Python que produz uma tabela com as temperatura em graus centígrados, equivalentes às temperaturas em graus Fahrenheit entre -40°F e 120°F .

4. Escreva um programa em Python que lê uma sequência de dígitos, sendo cada um do dígitos fornecido numa linha separada, e calcula o número inteiro composto por esses dígitos, pela ordem fornecida. Para terminar a sequência de dígitos é fornecido ao programa o inteiro -1 . Por exemplo, lendo os dígitos 1 5 4 5 8, o programa calcula o número inteiro 15458.

5. Escreva um programa em Python que lê um número inteiro positivo e calcula a soma dos seus dígitos pares. Por exemplo, Escreva um inteiro positivo

? 234567

Soma dos d´

ígitos pares:

6. Escreva um programa em Python que lê um número inteiro positivo e produz o número correspondente a inverter a ordem dos seus dígitos. Por exemplo,

Escreva um inteiro positivo

? 7633256

O número invertido é

6523367

68

CAPÍTULO 2. ELEMENTOS BÁSICOS DE PROGRAMAÇÃO

~
,

AO

7. Escreva um programa em Python que calcula o valor da série.

$1 + x + x^2 + x^3 +$

$2!$

$3!$

$\dots + x^n$

$n!$

para um dado valor de x e de n . O seu programa deve ter em atenção que cada termo da série pode ser obtido a partir do anterior, multiplicando-o por x/n .

Capítulo 3

Funções

oes

‘Can you do Addition?’ the White Queen asked. ‘What’s one and one and one and one and one and one and one and one and one and one?’

‘I don’t know,’ said Alice. ‘I lost count.’

‘She ca’n’t do Addition,’ the Red Queen interrupted. ‘Can you do Subtraction? Take nine from eight.’

‘Nine from eight I ca’n’t, you know,’ Alice replied very readily: ’but —’

‘She ca’n’t do Subtraction,’ said the White Queen. ‘Can you do Division? Divide a loaf by a knife — what’s the answer to that?’

Lewis Carroll, *Alice’s Adventures in Wonderland* No capítulo anterior considerámos apenas operações embutidas, as operações que já est’ao incluídas no Python. Como é evidente, não é possível que uma linguagem de programação forneça todas as operações que são necessárias para o desenvolvimento de uma certa aplicação. Por isso, durante o desenvolvimento de um programa, é necessário utilizar operações que não est’ao previamente definidas na linguagem de programação utilizada. As linguagens de programação fornecem mecanismos para a criação de novas operações e para a sua utilização num programa.

A possibilidade de agrupar informação e de tratar o grupo de informação como um todo, dando-lhe um nome que passará então a identificar o grupo, é parte fundamental da aquisição de conhecimento. Quando mais tarde se pretender utilizar a informação que foi agrupada e nomeada, ela poderá ser referida pelo

70

CAPÍTULO 3. FUNC

~

OES

seu nome, sem que seja preciso enumerar exaustivamente os pedaços individuais de informação que a constituem. Em programação, este aspecto está ligado aos conceitos de função, procedimento e subprograma. O Python utiliza a designação

“função”.

Antes de abordar a definição de funções em Python, recordemos o modo de definir e utilizar funções em matemática. Uma função (de uma variável) é um conjunto de pares ordenados que não contém dois pares distintos com o mesmo primeiro elemento. Ao conjunto de todos os primeiros elementos dos pares dá-se o nome de domínio da função, e ao conjunto de todos os segundos elementos dos pares dá-se

o nome de contradomínio da função. Por exemplo, o conjunto

$\{(1, 3), (2, 4), (3, 5)\}$ corresponde a uma função cujo domínio é $\{1, 2, 3\}$ e cujo contradomínio é $\{3, 4, 5\}$.

A definição de uma função listando todos os seus elementos corresponde à definição por extensão (ou enumeração).

Normalmente, as funções interessantes têm um número infinito de elementos, pelo que não é possível defini-las por extensão. Neste caso é necessário definir a função por compreensão (ou abstração), apresentando uma propriedade comum aos seus elementos. Um dos modos de definir uma função por compreensão corresponde a descrever o processo de cálculo dos segundos elementos dos pares a partir dos primeiros elementos dos pares, fornecendo uma expressão designatória em que a variável que nela aparece pertence ao domínio da função. Uma expressão designatória é uma expressão que se transforma numa designação quando a variável que nela aparece é substituída por uma constante.

Para calcular o valor da função para um dado elemento do seu domínio, basta substituir o valor deste elemento na expressão designatória que define a função.

Por exemplo, definindo a função natural de variável natural, f , pela expressão $f(x) = x * x$, estamos a definir o seguinte conjunto (infinito) de pares ordenados

$\{(1, 1), (2, 4), (3, 9), \dots\}$.

Note-se que $f(y) = y * y$ e $f(z) = z * z$ definem a mesma função (o mesmo conjunto de pares ordenados) que $f(x) = x * x$. As variáveis, tais como x , y e z nas expressões anteriores, que ao serem substituídas em todos os lugares que ocupam numa expressão dão origem a uma expressão equivalente, chamam-se variáveis mudas. Com a expressão $f(x) = x * x$, estamos a definir uma função cujo nome é f ; os elementos do domínio desta função (ou argumentos da função) são designados pelo nome x , e a regra para calcular o valor da função para um

3.1. DEFINIÇÃO

dado elemento do seu domínio corresponde a multiplicar esse elemento por si próprio.

Dada uma função, designa-se por conjunto de partida o conjunto a que pertencem os elementos do seu domínio, e por conjunto de chegada o conjunto a que pertencem os elementos do seu contradomínio. Dada uma função f e um elemento x pertencente ao seu conjunto de partida, se o par (x, y) pertence à função f , diz-se que o valor de $f(x)$ é y . É frequente que alguns dos elementos tanto do conjunto de chegada como do conjunto de partida da função não apareçam no conjunto de pares que correspondem à função. Se existe um valor z , pertencente ao conjunto de partida da função

ao, para o qual não existe nenhum

par cujo primeiro elemento é z , diz-se que a função é indefinida para z .

Note-se que a utilização de funções tem dois aspectos distintos: a definição da função e a aplicação da função.

1. A definição da função é feita fornecendo um nome (ou uma designação) para a função, uma indicação das suas variáveis (ou argumentos) e a indicação de um processo de cálculo para os valores da função, por exemplo, $f(x) = x * x$.

2. A aplicação da função é feita fornecendo o nome da função e um elemento do seu domínio para o qual se pretende calcular o valor, por exemplo, $f(5)$.

Analogamente ao processo de definição de funções em Matemática, em Python, o processo de utilização de funções compreende dois aspectos distintos: a definição da função, que, de um modo semelhante ao que se faz com a definição de funções em Matemática, é feita fornecendo o nome da função, os argumentos da função e um processo de cálculo para os valores da função (processo esse que é descrito por um algoritmo), e a aplicação da função a um valor, ou valores, do(s) seu(s) argumento(s). Esta aplicação corresponde à execução do algoritmo associado à função para valores particulares dos seus argumentos e em programação é vulgarmente designada por chamada à função.

3.1

Definiç~

ao de funç~

oes em Python

Para definir funç~oes em Python, é necessário indicar o nome da funç~ao, os seus argumentos (designados por parâmetros formais) e o processo de cálculo

72

CAPÍTULO 3. FUNC

OES

(algoritmo) dos valores da funç~ao (designado por corpo da funç~ao). Em notaç~ao BNF, uma funç~ao em Python é definida do seguinte modo1:

◆definiç~ao de funç~ao◆ ::= def ◆nome◆ (◆parâmetros formais◆):
CR

TAB+ ◆corpo◆ TAB-

◆parâmetros formais◆ ::= ◆nada◆ ◆nomes◆

◆nomes◆ ::= ◆nome◆

◆nome◆, ◆nomes◆

◆nada◆ ::=

◆corpo◆ ::= ◆definiç~ao de funç~ao◆* ◆instruç~oes em funç~ao◆

◆instruç~oes em funç~ao◆ ::= ◆instruç~ao em funç~ao◆ CR

◆instruç~ao em funç~ao◆ CR ◆instruç~oes em funç~ao◆

◆instruç~ao em funç~ao◆ ::= ◆instruç~ao◆

◆instruç~ao return◆

◆instruç~ao return◆ ::= return

return ♦expressão♦

Nestas expressões, símbolo não terminal ♦parâmetros formais♦, correspondente a zero ou mais nomes, especifica os parâmetros da função e o símbolo não terminal

♦corpo♦ especifica o algoritmo para o cálculo do valor da função. No ♦corpo♦ de uma função pode ser utilizada uma instrução adicional, a instrução return.

Para já, não vamos considerar a possibilidade de utilizar a ♦definição de função♦

dentro do corpo de uma função.

Por exemplo, em Python, a função, quadrado, para calcular o quadrado de um número arbitrário, é definida do seguinte modo: def quadrado (x):

```
return x * x
```

Nesta definição:

1. O nome x representa o argumento da função, o qual é designado por parâmetro formal. O parâmetro formal vai indicar o nome pelo qual o argumento da função é representado dentro do corpo da função; 2. A instrução return x * x corresponde ao corpo da função.

1A ♦definição de função♦ corresponde em Python a uma ♦definição♦.

3.2. APLICAC

~
s

AO DE FUNC

~
s

OES EM PYTHON

A semântica da definição de uma função é a seguinte: ao encontrar a definição de uma função, o Python cria um nome, correspondente ao nome da função, e associa esse nome com um processo de cálculo para os argumentos da função.

Ao definir uma função, o Python adiciona ao ambiente o nome correspondente ao nome da função, associando-o ao algoritmo para calcular o valor da função.

Consideremos a seguinte interação:

```
> > > quadrado
```

```
NameError: name 'quadrado' is not defined
```

```
> > > def quadrado (x):
```

```
...
```

```
    return x * x
```

```
...
```

```
> > > quadrado
```

```
<function quadrado at 0x10f4618>
```

Na primeira linha, ao fornecermos ao Python o nome quadrado este indica-nos que não conhece o nome. Nas três linhas seguintes, definimos a função quadrado. Recorde-se que as linhas que apresentam ... como carácter de pronto correspondem a linhas de continuação da definição da função e uma linha em branco indica ao Python que terminámos a sua definição. Quando, na linha seguinte fornecemos ao Python o nome quadrado este indica-nos que quadrado corresponde a uma função e qual a posição de memória em que esta se encontra armazenada.

3.2

Aplicação

ao de função

oes em Python

Uma vez definida, uma função pode ser usada do mesmo modo que as funções embutidas, fornecendo ao Python, numa expressão, o nome da função seguido do número apropriado de argumentos (os parâmetros concretos). A partir do momento em que uma função é definida, passa a existir em Python uma nova expressão, correspondente a uma aplicação de função (ver a definição apresentada na página 33), a qual é definida do seguinte modo:

$\diamond \text{aplica\~ao de fun\~ao} \diamond ::= \diamond \text{nome} \diamond (\diamond \text{par\~ametros concretos} \diamond)$

$\diamond \text{par\~ametros concretos} \diamond ::= \diamond \text{nada} \diamond \diamond \text{express\~oes} \diamond$

74

CAPÍTULO 3. FUNC

~
s

OES

$\diamond \text{express\~oes} \diamond ::= \diamond \text{express\~ao} \diamond$

$\diamond \text{express\~ao} \diamond, \diamond \text{express\~oes} \diamond$

Nesta definiç~ao, $\diamond \text{nome} \diamond$ corresponde ao nome da funç~ao e o número de express~oes em $\diamond \text{par\~ametros concretos} \diamond$ é igual ao número de parâmetros formais da funç~ao.

Este último aspecto n~ao pode ser definido em notaç~ao BNF. Quando uma funç~ao é aplicada é comum dizer-se que a funç~ao foi chamada.

Usando a funç~ao quadrado podemos originar a interacç~ao:

> > > quadrado(7)

49

> > > quadrado(2, 3)

TypeError: quadrado() takes exactly 1 argument (2 given) A interacç~ao anterior mostra que se fornecermos à funç~ao quadrado o argumento 7, o valor da funç~ao é 49. Se fornecermos dois argumentos à funç~ao, o Python gera um erro indicando-nos que esta funç~ao recebe apenas um argumento.

Consideremos, de um modo mais detalhado, os passos seguidos pelo Python ao calcular o valor de uma funç~ao. Para calcular o valor da aplicaç~ao de uma funç~ao, o Python utiliza a seguinte regra:

1. Avalia os parâmetros concretos (por qualquer ordem).

2. Associa os parâmetros formais da funç~ao com os valores dos parâmetros concretos calculados no passo anterior. Esta associaç~ao é feita com base na posiç~ao dos parâmetros, isto é, o primeiro parâmetro concreto é associado ao primeiro parâmetro formal, e assim

sucessivamente.

3. Cria um novo ambiente, o ambiente local à função, definido pela associação entre os parâmetros formais e os parâmetros concretos. No ambiente local executa as instruções correspondentes ao corpo da função. O ambiente local apenas existe enquanto a função estiver a ser executada e é apagado pelo Python quando termina a execução da função.

Com a definição de funções, surge uma nova instrução em Python, a instrução `return`, a qual apenas pode ser utilizada dentro do corpo de uma função, e cuja sintaxe foi definida na página 72. Ao encontrar a instrução `return`, o Python calcula o valor da expressão associada a esta instrução (se ela existir),



3.2. APLICAC

~
,

AO DE FUNC

~
,

OES EM PYTHON

75

Ambiente global

quadrado

função

x

5

y

7

Figura 3.1: Ambiente com os nomes `quadrado`, `x` e `y`.

termina a execução do corpo da função, sendo o valor da função o valor desta expressão. Este valor é normalmente designado pelo valor

devolvido pela função.

Se a instrução `return` não estiver associada a uma expressão, então a função não devolve nenhum valor (note-se que já vimos, na página 56, que podem existir funções que não devolvem nenhum valor). Por outro lado, se todas as instruções correspondentes ao corpo da função forem executadas sem encontrar uma instrução `return`, a execução do corpo da função termina e a função não devolve nenhum valor.

Consideremos agora a seguinte interação, efectuada depois da definição da função `quadrado`:

```
> > > x = 5
```

```
> > > y = 7
```

```
> > > quadrado(y)
```

49

As duas primeiras linhas dão origem ao ambiente apresentado na Figura 3.1

(neste ambiente, o nome `quadrado` já era conhecido, estando associado com o processo de cálculo desta função). Ao encontrar a expressão `quadrado(y)` na terceira linha da nossa interação, o Python calcula o valor de `y` e cria um novo ambiente, representado na Figura 3.2 com o título *Ambiente local a quadrado*, no qual a variável `x`, o parâmetro formal de `quadrado`, está associada ao valor do parâmetro concreto 7. É então executado o corpo da função `quadrado`, o qual apenas contém uma instrução `return x x`. A *semântica da instrução return* leva à avaliação da expressão `x x`, cujo valor é 49. Este valor corresponde ao valor da aplicação função `quadrado(y)`. A instrução `return` tem também como efeito fazer desaparecer o ambiente que foi criado com a avaliação da função, pelo quando o valor 49 é devolvido o Python volta a ter o ambiente apresentado na Figura 3.1.



76

CAPÍTULO 3. FUNC

OES

Ambiente global

quadrado

função

x

5

y

7

Ambiente local a quadrado

x

7

Figura 3.2: Ambiente criado durante a avaliação de quadrado(y).

Podemos então dizer que quando uma função é chamada, é criado um ambiente local, o qual corresponde a uma associação entre os parâmetros formais e os parâmetros concretos. Este ambiente local desaparece no momento em que termina a avaliação da função que deu origem à sua criação.

3.3

Abstracção

ao procedimental

A criação de funções corresponde a uma forma de abstracção, a qual consiste em nomear uma sequência de operações que permitem atingir um determinado objectivo (a definição da função). Uma vez definida uma função, para a aplicar a determinados parâmetros concretos basta escrever o seu nome seguido desses parâmetros (a aplicação da função), tal como se de uma função embutida se tratasse. A introdução dos parâmetros formais permite que uma função represente um número potencialmente infinito de operações (idênticas), dependendo apenas dos parâmetros concretos que lhe são fornecidos.

A abstracção procedimental consiste em dar um nome à sequência de acções que serve para atingir um objectivo (e consequentemente abstrair do modo como as funções realizam as suas tarefas), e em utilizar esse nome, sempre que desejarmos atingir esse objectivo sem termos de considerar explicitamente cada uma das acções individuais que a constituem.

A associação de um nome a uma sequência de acções, a qual é referida pelo seu nome em vez de enumerar todas as instruções que a constituem, é por nós utilizada diariamente. Suponhamos que desejamos que alguém abra uma determinada porta. Ao dizermos “abra a porta”, estamos implicitamente a referir

3.3. ABSTRACC

~
,

AO PROCEDIMENTAL

77

uma determinada sequência de instruções: “mova-se em direcção à porta, de-termine para onde a porta se abre, rode o manípulo, empurre ou puxe a porta, etc.”. Sem a capacidade de nomear sequências de instruções, seria impossível comunicarmos, uma vez que teríamos de explicitar tudo ao mais ínfimo pormenor.

A abstracção procedimental consiste em abstrair do modo como as funções realizam as suas tarefas, concentrando-se apenas na tarefa que as funções realizam.

Ou seja, a separação do “como” de “o que”.

A abstracção procedimental permite-nos desenvolver programas complexos, ma-nipulando entidades complexas, sem nos perdermos em pormenores em relação à especificação minuciosa do algoritmo que manipula essas entidades. Usando a abstracção procedimental consideramos funções como caixas pretas – sabemos o que eles fazem, mas não nos interessa saber como o fazem. Para ilustrar este aspecto, recordemos que a função quadrado foi definida do seguinte modo: def quadrado (x):

```
return x * x
```

No entanto, para a finalidade da sua utilização, os pormenores da realização da função quadrado são irrelevantes. Poderíamos ter

definido a função quadrado do seguinte modo (esta função calcula o quadrado de um número somando o número consigo próprio um número de vezes igual ao número): def quadrado (x):

```
res = 0
```

```
cont = 0
```

```
while cont != x:
```

```
res = res + x
```

```
cont = cont + 1
```

```
return res
```

e obtemos exactamente o mesmo comportamento. O que nos interessa saber, quando usamos a função quadrado, é que esta recebe como argumento um número e produz o valor do quadrado desse número – ou seja, interessa-nos saber o que esta função faz, como o faz não nos interessa (Figura 3.3). É a isto que corresponde a abstracção procedimental.

78

CAPÍTULO 3. FUNC



Figura 3.3: O conceito de caixa preta.

A abstracção procedimental permite-nos considerar as funções como ferramentas que podem ser utilizadas na construção de outras funções. Ao considerarmos uma destas ferramentas, sabemos quais os argumentos que recebe e qual o resultado produzido, mas não sabemos (ou não nos preocupamos em saber) como ela executa a sua tarefa — apenas em que consiste essa tarefa.

3.4

Exemplos

3.4.1

Nota final de uma disciplina

Suponhamos que numa determinada disciplina existiam três provas de avaliação, e que a nota da disciplina era calculada como a média aritmética das três classificações obtidas.

Pretendemos escrever uma função que recebe três argumentos, correspondentes às notas das provas realizadas, e que devolve a classificação final obtida na disciplina. Se a média das provas realizadas for inferior a 9.5 a classificação será “Reprovado”, em caso contrário a classificação será o número inteiro que corresponde ao arredondamento da média da disciplina.

A seguinte função realiza o cálculo da nota final da disciplina: `def calcula_nota (n1, n2, n3):`

```
media = (n1 + n2 + n3) / 3
```

```
if media < 9.5:
```

```
    return 'Reprovado'
```

```
else:
```

```
    return round(media)
```

Com esta função podemos gerar a interação:

```
> > > calcula_nota(14, 16, 13)
```

3.4. EXEMPLOS

```
79
```

```
14
```

```
> > > calcula_nota(3, 9, 12)
```

```
'Reprovado'
```

Com esta função, podemos escrever o seguinte programa: `def`

```
calcula_nota (n1, n2, n3):
```

```
media = (n1 + n2 + n3) / 3
```

```
if media < 9.5:
```

```
    return 'Reprovado'
```

```
else:
```

```
    return round(media)
```

```
num = input('Qual o número do aluno\n(zero para terminar)\n? ')
```

```
while num != '0':
```

```
    n_1, n_2, n_3 = eval(input('Quais as suas notas\n? '))
    print('A nota obtida pelo aluno ', num, \
```

```
        ' é ', calcula_nota(n_1, n_2, n_3))
```

```
num = input('Qual o número do aluno\n(zero para terminar)\n? ')
```

Recorde-se da página 56, que um programa é constituído por zero ou mais definições seguidas de uma ou mais instruções. No nosso caso, o programa é constituído pela definição da função calcula_nota, seguida das instruções que, através de um ciclo, calculam as notas de vários alunos. Note-se que a função print ocupa duas linhas, terminando a primeira com o símbolo “\”. Este símbolo indica ao Python que a linha continua na linha seguinte. Com este programa podemos obter a interação:

Qual o número do aluno

(zero para terminar)

? 12345

Quais as suas notas

? 12, 15, 19

A nota obtida pelo aluno

12345

é

15

Qual o número do aluno

(zero para terminar)

? 12346

80

CAPÍTULO 3. FUNC

~
,

OES

Quais as suas notas

? 8, 10, 6

A nota obtida pelo aluno

12346

é

Reprovado

Qual o número do aluno

(zero para terminar)

? 0

> > >

3.4.2

Potência

Uma operação comum em Matemática é o cálculo da potência de um número.

Dado um número qualquer, x (a base), e um inteiro não negativo, n (o expoente), define-se potência da base x ao expoente n , escrito x^n , como sendo o produto de x por si próprio n vezes. Por convenção, $x^0 = 1$.

Com base nesta definição podemos escrever a seguinte função em Python: `def potencia (x , n):`

```
res = 1

while n > 0:

    res = res * x

    n = n - 1

return res
```

Com esta função, podemos gerar a seguinte interação:

```
> > > potencia(3, 2)
```

```
9
```

```
> > > potencia(2, 8)
```

```
256
```

```
> > > potencia(3, 100)
```

```
515377520732011331036461129765621272702107522001
```

3.4. EXEMPLOS

```
81
```

3.4.3

Factorial

Em matemática, a função factorial, de um inteiro não negativo n , representada por $n!$, é o produto de todos os inteiros positivos menores ou iguais a n . Por exemplo, $5! = 5 * 4 * 3 * 2 * 1 = 120$. Por convenção, o valor de $0!$ é 1.

Com base na definição anterior, podemos escrever a seguinte função em Python para calcular o factorial de um inteiro::

```
def factorial (n):
```

```
    fact = 1
```

```
    while n > 0:
```

```
        fact = fact * n
```

```
n = n - 1
```

```
return fact
```

Com esta função obtemos a interacção:

```
> > > factorial(3)
```

```
6
```

```
> > > factorial(21)
```

```
51090942171709440000
```

3.4.4

Máximo divisor comum

O máximo divisor comum entre dois inteiros m e n diferentes de zero, escrito $\text{mdc}(m, n)$, é o maior inteiro positivo p tal que tanto m como n são divisíveis por p . Esta descrição define uma função matemática no sentido em que podemos reconhecer quando um número é o máximo divisor comum de dois inteiros, mas, dados dois inteiros, esta definição não nos indica como calcular o seu máximo divisor comum. Este é um dos aspectos em que as funções em informática diferem das funções matemáticas. Embora tanto as funções em informática como as funções matemáticas especifiquem um valor que é determinado por um 2Segundo [Biggs, 1979], a função factorial já era conhecida por matemáticos indianos no início do Século XII, tendo a notação

ao $n!$ sido introduzida pelo matemático francês Christian Kramp (1760–1826) em 1808.

82

CAPÍTULO 3. FUNC

~
s

OES

ou mais parâmetros, as funções em informática devem ser eficazes, no sentido de que têm de especificar, de um modo claro, como calcular os valores.

Um dos primeiros algoritmos a ser formalizado corresponde ao cálculo

do máximo divisor comum entre dois inteiros e foi enunciado, cerca de 300 anos a.C., por Euclides no seu Livro VII. A descrição originalmente apresentada por Euclides é complicada (foi enunciada há 2300 anos) e pode ser hoje expressa de um modo muito mais simples:

1. O máximo divisor comum entre um número e zero é o próprio número.
2. Quando dividimos um número por um menor, o máximo divisor comum entre o resto da divisão e o divisor é o mesmo que o máximo divisor comum entre o dividendo e o divisor.

Com base na descrição anterior, podemos enunciar o seguinte algoritmo para calcular o máximo divisor comum entre m e n , $\text{mdc}(m, n)$, utilizando o algoritmo de Euclides:

1. Se $n = 0$, então o máximo divisor comum corresponde a m . Note-se que isto corresponde ao facto de o máximo divisor comum entre um número e zero ser o próprio número.
2. Em caso contrário, calculamos o resto da divisão entre m e n , o qual utilizando a função embutida do Python (ver Tabela 2.2) será dado por $r = m \% n$ e repetimos o processo com m igual ao valor de n (o divisor) e n igual ao valor de r (o resto da divisão).

Os passos descritos serão repetidos enquanto n não for zero. Na Tabela 3.1

apresentamos os passos seguidos no cálculo do máximo divisor comum entre 24

e 16.

Podemos então escrever a seguinte função em Python para calcular o máximo divisor comum entre os inteiros m e n :

```
def mdc (m, n):
```

```
    while n != 0:
```

```
        resto = m % n
```

```
        m = n
```

3.4. EXEMPLOS

83

m

n

m % n

24 16

8

16

8

0

8

0

8

Tabela 3.1: Passos no cálculo do máximo divisor comum entre 24 e 16.

n = resto

return m

Uma vez definida a função mdc, podemos gerar a seguinte interação:

```
> > > mdc(24, 16)
```

8

```
> > > mdc(16, 24)
```

8

```
> > > mdc(35, 14)
```

7

3.4.5

Raiz quadrada

Vamos supor que queríamos escrever uma função em Python para calcular a raiz quadrada de um número positivo. Como poderíamos calcular o valor de

$\sqrt{x}$ para um dado x ?

Comecemos por considerar a definição matemática de raiz quadrada: $\sqrt{x}$ é o y tal que $y^2 = x$. Esta definição, típica da Matemática, diz-nos o que é uma raiz quadrada, mas não nos diz nada sobre o processo de cálculo de uma raiz quadrada. Com esta definição, podemos determinar se um número corresponde à raiz quadrada de outro, podemos provar propriedades sobre as raízes quadradas mas não temos qualquer pista sobre o modo de calcular raízes quadradas.

Iremos utilizar um algoritmo para o cálculo de raízes quadradas, que foi introduzido no início da nossa era pelo matemático grego Heron de Alexandria (c.

10–75 d.C.)³. Este algoritmo, corresponde a um método de iterações sucessivas, 3Ver [Kline, 1972].

84

CAPÍTULO 3. FUNC

, ~

OES

Número

Aproximaç~

ao

Nova aproximaç~

ao

$\sqrt{}$

da tentativa

para

2

$1 + 2$

0

1
 $1 = 1.5$
 2
 $1.5 + 2$
 1
 1.5
 $1.5 = 1.4167$
 2
 $1.4167 +$
 2
 2
 1.4167
 $1.4167 = 1.4142$
 2
 3
 1.4142
 ...

$\sqrt{}$

Tabela 3.2: Sucessivas aproximações para 2.

em que a partir de um “palpite” inicial para o valor da raiz quadrada de x , digamos p_0 , nos permite melhorar sucessivamente o valor aproximado de $\sqrt{x}$.

Em cada iteração, partindo de um valor aproximado, p_i , para a raiz quadrada de x podemos calcular uma aproximação melhor, p_{i+1} , para a raiz quadrada de x , através da seguinte fórmula:

$$p_i + x$$

p

p_i

$i + 1 =$

2

.

$\sqrt{}$

$\sqrt{}$

Suponhamos, por exemplo, que desejamos calcular $\sqrt{2}$. Sabemos que $\sqrt{2}$ é um v´irgula qualquer coisa. Seja entˆao 1 o nosso palpite inicial, $p_0 = 1$. A Tabela 3.2

$\sqrt{}$

mostra-nos a evoluç˜ao das primeiras aproximaç˜oes calculadas para $\sqrt{2}$.

Um aspecto que distingue este exemplo dos apresentados nas secç˜oes anteriores ´e o facto de o c´alculo da raiz quadrada ser realizado por um m´etodo aproximado.

Isto significa que n˜ao vamos obter o valor exacto da raiz quadrada⁴, mas sim uma aproximaç˜ao que seja suficientemente boa para o fim em vista.

Podemos escrever a seguinte funç˜ao em Python que corresponde ao algoritmo apresentado:

```
def calcula_raiz(x, palpite):  
    while not bom_palpite(x, palpite):  
        palpite = novo_palpite(x, palpite)  
    return palpite
```

Esta funç˜ao ´e bastante simples: fornecendo-lhe um valor para o qual calcular a raiz quadrada (x) e um palpite ($palpite$), enquanto estivermos perante um ⁴Na realidade, este ´e um n´umero irracional.

3.4. EXEMPLOS

palpite que não é bom, deveremos calcular um novo palpite; se o palpite for um bom palpite, esse palpite será o valor da raiz quadrada.

Esta função pressupõe que existem outras funções para decidir se um dado palpite para o valor da raiz quadrada de x é bom (a função bom palpite) e para calcular um novo palpite (a função novo palpite). A isto chama-se pensamento positivo! Note-se que esta técnica de pensamento positivo nos permitiu escrever uma versão da função calcula raiz em que os problemas principais estão identificados, versão essa que é feita em termos de outras funções. Para podermos utilizar a função calcula raiz teremos de desenvolver funções para decidir se um dado palpite é bom e para calcular um novo palpite.

Esta abordagem do pensamento positivo corresponde a um método largamente difundido para limitar a complexidade de um problema, a que se dá o nome de abordagem do topo para a base⁵. Ao desenvolver a solução de um problema utilizando a abordagem do topo para a base, começamos por dividir esse problema noutros mais simples. Cada um destes problemas recebe um nome, e é então desenvolvida uma primeira aproximação da solução em termos dos principais subproblemas identificados e das relações entre eles. Seguidamente, aborda-se a solução de cada um destes problemas utilizando o mesmo método. Este processo termina quando se encontram subproblemas para os quais a solução é trivial.

O cálculo do novo palpite é bastante fácil, bastando recorrer à fórmula apresentada pelo algoritmo, a qual é traduzida pela função: def novo_palpite (x , palpite):

```
return (palpite +  $x$  / palpite) / 2
```

Resta-nos decidir quando estamos perante um bom palpite. É evidente que a satisfação com dado palpite vai depender do objectivo para o qual estamos a calcular a raiz quadrada: se estivermos a fazer cálculos que exijam alta precisão, teremos de que exigir uma maior precisão do que a que consideraríamos em cálculos grosseiros.

A definição matemática de raiz quadrada, $\sqrt{x}$ é o y tal que $y^2 = x$, dá-nos uma boa pista para determinar se um palpite p_i é ou não bom. Para que o palpite p_i seja um bom palpite, os valores $(p_i)^2$ e x deverão ser suficientemente próximos. Uma das medidas utilizadas em Matemática para decidir se dois

⁵Do inglês, “top-down design”.

OES

números são “quase iguais”, chamada erro absoluto, corresponde a decidir se o valor absoluto da sua diferença é menor do que um certo limiar⁶. Utilizando esta medida, diremos que π é uma boa aproximação de $\sqrt{x}$ se $|\pi^2 - x| < \delta$, em que δ é um valor suficientemente pequeno. Supondo que δ corresponde ao nome `delta`, o qual define o limiar de aproximação exigido pela nossa aplicação, podemos escrever a função:

```
def bom_palpite (x, palpite):
```

```
    return abs(x - palpite * palpite) < delta
```

Note-se que poderíamos ser tentados a escrever a seguinte função para decidir se uma dada aproximação é um bom palpite:

```
def bom_palpite (x, palpite):
```

```
    if abs(x - palpite * palpite) < delta:
```

```
        return True
```

```
    else:
```

```
        return False
```

Esta função avalia a expressão `abs(x - palpite * palpite) < delta`, devolvendo `True` se esta for verdadeira e `False` se a expressão for falsa. Note-se no entanto que o valor devolvido pela função é exactamente o valor da expressão `abs(x - palpite * palpite) < delta`, daí a simplificação que introduzimos na nossa primeira função.

Com as funções apresentadas, e definindo `delta` como sendo 0.0001, podemos gerar a interacção

```
> > > delta = 0.0001
```

```
> > > calcula_raiz(2, 1)
```

```
1.4142156862745097
```

Convém notar que o cálculo da raiz quadrada através da função `calcula_raiz` não é natural, pois obriga-nos a fornecer um palpite (o

que não acontece quando calculamos a raiz quadrada com uma calculadora). Sabendo que 1 pode ser o palpite inicial para o cálculo da raiz quadrada de qualquer número, e que apenas 6 Uma outra alternativa corresponde a considerar o erro relativo, o quociente entre o erro absoluto e o valor correcto, $|(pi)^2 - x|/x$.

3.4. EXEMPLOS

87

podemos calcular raízes de números não negativos, podemos finalmente escrever a função raiz que calcula a raiz quadrada de um número não negativo: `def raiz (x):`

```
delta = 0.001
```

```
if x >= 0:
```

```
    return calcula_raiz (x, 1)
```

```
else:
```

```
    raise ValueError ('raiz: argumento negativo')
```

 Na função raiz usámos a instrução `raise` que força a geração de um erro de execução. A razão da nossa decisão de gerar um erro em lugar de recorrer à geração de uma mensagem através da função `print` resulta do facto que quando fornecemos à função raiz um argumento negativo não queremos que a execução desta função continue, alertando o utilizador que algo de errado aconteceu.

A instrução `raise` tem a seguinte sintaxe em notação BNF:

❖instrução raise❖ ::= raise ❖nome❖ (❖mensagem❖)

❖mensagem❖ ::= ❖cadeia de caracteres❖

O símbolo não terminal ❖nome❖ foi definido na página 72. A utilização de ❖nome❖

nesta instrução diz respeito a nomes que correspondem à identificação de vários tipos de erros que podem surgir num programa, alguns dos quais são apresentados na Tabela 3.3. Note-se que várias das situações correspondentes a erros apresentados na Tabela 3.3 ainda não foram apresentadas neste livro. A

❖mensagem❖ corresponde à mensagem que é mostrada pelo Python com a indicação do erro.

Assim, com a função raiz, podemos obter a seguinte interação:

```
> > > raiz(2)
```

```
1.4142156862745097
```

```
> > > raiz(-1)
```

ValueError: raiz: argumento negativo

A função raiz é a nossa primeira função definida à custa de um certo número de outras funções. Cada uma destas funções aborda um problema específico:

88

CAPÍTULO 3. FUNC

~
,

OES

Nome

Situaç~

ao correspondente ao erro

AttributeError

Referência a um atributo n~

ao existente num objecto.

ImportError

Importaç~

ao de uma biblioteca n~

ao existente.

IndexError

Erro gerado pela referência a um índice fora da gama de um tuplo ou de uma lista.

KeyError

Referência a uma chave inexistente num dicionário.

`NameError`

Referência a um nome que não
existe.

`SyntaxError`

Erro gerado quando uma das funções
eval ou input
encontram uma expressão
com a sintaxe incorrecta.

`ValueError`

Erro gerado quando uma função
recebe um
argumento de tipo correcto mas cujo valor não
é
apropriado.

`ZeroDivisionError`

Erro gerado pela divisão
por zero.

Tabela 3.3: Alguns dos identificadores de erros em Python.

raiz

calcula raiz





bom palpite

novo palpite

Figura 3.4: Funções associadas ao cálculo da raiz.

como determinar se um palpite é suficientemente bom; como calcular um novo palpite; *etc.* Podemos olhar para a função raiz como sendo definida através de um agrupamento de outras funções (Figura 3.4), o que corresponde à abstracção procedural.

Sob esta perspectiva, a tarefa de desenvolvimento de um programa corresponde a definir várias funções, cada uma resolvendo um dos subproblemas do problema a resolver, e “ligá-las entre si” de modo apropriado.

Antes de terminar esta secção convém discutir a relação entre funções matemáticas e as funções em Python que correspondem a essas funções. As funções que temos definido até aqui comportam-se basicamente como funções matemáticas: ao receberem um valor correspondente a um argumento do seu domínio, devolvem o valor correspondente da função. No entanto, as funções em Python devem estar associadas a um algoritmo que calcula o valor da função, ao passo que no caso das funções matemáticas basta definir o que elas são. Em

3.4. EXEMPLOS

Número de termos

Aproximaç~

ao a $\sin(1.57)$

1

1.57

2

$1.57 - 1.573 = 0.92501$

3!

3

$1.57 - 1.573 + 1.575 = 1.00450$

3!

5!

4

$1.57 - 1.573 + 1.575 - 1.577 = 0.99983$

3!

5!

7!

Tabela 3.4: Sucessivas aproximações ao cálculo de $\sin(1.57)$.

resumo, a informática baseia-se em conhecimento procedimental, ou seja como fazer as coisas, ao passo que a matemática se baseia em conhecimento declarativo, ou seja, o que as coisas são.

3.4.6

Seno

Apresentamos um procedimento para calcular o valor do seno. Este exemplo partilha com o cálculo da raiz o facto de ser realizado por um método aproximado.

O seno é uma função trigonométrica. Dado um triângulo rectângulo com um de seus ângulos internos igual α , define-se $\sin(\alpha)$ como sendo a razão entre o cateto oposto e a hipotenusa deste triângulo. Para calcular o valor de seno podemos utilizar o desenvolvimento em série de Taylor o qual fornece o valor de $\sin(x)$ para um valor de x em radianos:

$$\infty$$



$$\sin(x) =$$

$$(-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

$$n=0$$

$$n=0$$

A série anterior fornece-nos um processo de cálculo para o $\sin(x)$, mas, aparentemente, não nos serve de muito, uma vez que, para calcular o valor da função para um dado argumento, teremos de calcular a soma de um número infinito de termos. No entanto, podemos utilizar a fórmula anterior para calcular uma aproximação à função seno, considerando apenas um certo número de termos.

Na Tabela 3.4 apresentamos alguns exemplos de aproximações ao cálculo de $\sin(1.57)$ (note-se que $1.57 \approx \pi/2$, cujo valor do seno é 1).

Usando uma técnica semelhante à da secção anterior, consideremos o seguinte

$$90$$

CAPÍTULO 3. FUNC

$$\sim$$

OES

algoritmo para calcular $\sin(x)$, o qual começa com uma aproximação ao valor do seno (correspondente ao primeiro termo da série): 1. Se o termo a adicionar for suficientemente pequeno, então a aproximação considerada será o valor da série.

2. Em caso contrário, calculamos uma aproximação melhor, por adição de mais um termo da série.

Com base no que dissemos, podemos esboçar uma função em Python para calcular o seno:

```
def sin (x):  
  
    n = 0  
  
    # termo da série em consideraç~  
  
    ao  
  
    termo = calc_termo(x, n)  
  
    seno = termo  
  
    while not suf_pequeno(termo):  
  
        n = n + 1  
  
        termo = calc_termo(x, n)  
  
        seno = seno + termo  
  
    return seno
```

Esta função pressupõe que existem outras funções para decidir se um termo é suficientemente pequeno (a função `suf_pequeno`) e para calcular o termo da série que se encontra na posição n (a função `calc_termo`). Com base na discussão apresentada na secção anterior, a função `suf_pequeno` é definida trivialmente do seguinte modo:

```
def suf_pequeno (valor):  
  
    return abs(valor) < delta  
  
delta = 0.0001
```

Para calcular o termo na posição n da série, notemos que o factor $(-1)^n$, em cada um dos termos na série de Taylor, na realidade define qual o sinal do termo: o sinal é positivo se n for par, e é negativo em caso contrário. Por esta razão, a função `sinal` não é definida em termos de potências mas sim em termos da

3.5. ESTRUTURAC

OES

91

paridade de n . Na segunda linha da função `calc_termo` usamos novamente o símbolo “\”, o símbolo de continuação, que indica ao Python que a linha que estamos escrevendo continua na próxima linha. A utilização do símbolo de continuação não é obrigatória, servindo apenas para tornar o programa mais fácil de ler.

```
def calc_termo (x, n):
```

```
    return sinal(n) * \
```

```
    potencia(x, 2 * n + 1) / factorial(2 * n + 1)
```

```
def sinal (n):
```

```
    if n % 2 == 0:
```

```
        # n é par
```

```
        return 1
```

```
    else:
```

```
        return -1
```

Com estas funções podemos gerar a interação:

```
>>> sin(1.57)
```

```
0.99999996270418701
```

```
3.5
```

Estruturação

ao de função

oes

Na Seção 3.1 estudamos o modo de definir funções em Python. Nesta seção, apresentamos o modo de criar uma estrutura nas funções e analisamos as consequências da criação dessa estrutura.

Consideremos a função potencia apresentada na Secção 3.4.2: def potencia (x , n):

```
res = 1
```

```
while n > 0:
```

```
res = res * x
```

```
n = n - 1
```

```
return res
```

7A função potencia foi definida na Secção 3.4.2 e a função factorial na Secção 3.4.3.

92

CAPÍTULO 3. FUNC

~
,

OES

É fácil constatar que esta função apenas produz resultados para valores do expoente que sejam inteiros positivos ou nulos. Se fornecermos à função um valor negativo para o expoente, o Python gera um ciclo infinito, pois a condição do ciclo while, $n > 0$, nunca é satisfeita. Isto corresponde a uma situação que queremos obviamente evitar. Tendo em atenção que $b^{-e} = 1/b^e$, podemos pensar na seguinte solução que lida tanto com o expoente positivo como com o expoente negativo:

```
def potencia (x, e):
```

```
res = 1
```

```
if e >= 0:
```

```
while e > 0:
```

```
res = res * x
```

```
e = e -1
```

```
return res
```

```
else:
```

```
while e < 0:
```

```
res = res / x
```

```
e = e + 1
```

```
return res
```

Com esta nova função podemos gerar a interação:

```
> > > potencia(3, 2)
```

```
9
```

```
> > > potencia(3, -2)
```

```
0.11111111111111111
```

```
> > > potencia(3, 0)
```

```
1
```

A desvantagem desta solução é a de exigir dois ciclos muito semelhantes, um que trata o caso do expoente positivo, multiplicando sucessivamente o resultado por x, e outro que trata o caso do expoente negativo, dividindo sucessivamente o resultado por x.

O Python apresenta uma alternativa para abordar problemas deste tipo, baseada no conceito de estrutura de blocos. A estrutura de blocos é muito importante em programação, existindo uma classe de linguagens de programação, chamadas

3.5. ESTRUTURAC

```
~  
,
```

AO DE FUNC

```
~  
,
```

OES

93

linguagens estruturadas em blocos, que são baseadas na definição de blocos.

Historicamente, a primeira linguagem desta classe foi o Algol, desenvolvida na década de 50, e muitas outras têm surgido, o Python é uma destas linguagens.

A ideia subjacente à estrutura de blocos consiste em definir funções dentro das quais existem outras funções. Em Python, qualquer função pode ser considerada como um bloco, dentro da qual podem ser definidos outros blocos.

Nas linguagens estruturadas em blocos, toda a informação definida dentro de um bloco pertence a esse bloco, e só pode ser usada por esse bloco e pelos blocos definidos dentro dele. Esta regra permite a protecção efectiva da informação definida em cada bloco da utilização não autorizada por parte de outros blocos.

Podemos pensar nos blocos como sendo egoístas, não permitindo o acesso à sua informação pelos blocos definidos fora deles.

A razão por que os blocos não podem usar a informação definida num bloco interior é que essa informação pode não estar pronta para ser utilizada até que algumas acções lhe sejam aplicadas ou pode exigir que certas restrições sejam verificadas. O bloco onde essa informação é definida é o único que sabe quais condições de utilização dessa informação e, consequentemente, o único que pode garantir que essa sequência de acções é aplicada antes da informação ser usada.

Recorde-se da página 72 que uma função em Python foi definida do seguinte modo:

```
definição de função ::= def nome (parâmetros formais):  
CR
```

```
TAB corpo TAB-
```

```
corpo ::= definição de função * instruções +
```

Até agora, o corpo de todas as nossas funções tem sido apenas constituído por instruções e o componente opcional definição de função não tem existido. No entanto, são as definições de funções dentro do corpo que permitem a definição da estrutura de blocos num programa em Python: cada função definida no corpo de uma outra função corresponde a um bloco.

Recorrendo a esta definição mais alargada de uma função, podemos escrever uma nova versão da função potencia:

```
def potencia (x , n):
```

```
def potencia_aux (b, e):
```

```
94
```

CAPÍTULO 3. FUNC

~
,

OES

```
res = 1
```

```
while e > 0:
```

```
res = res * x
```

```
e = e - 1
```

```
return res
```

```
if n >= 0:
```

```
return potencia_aux(x, n)
```

```
else:
```

```
return 1 / potencia_aux(x, -n)
```

O corpo desta função contém a definição de uma função, potencia aux, e uma instrução if. Com esta função, podemos gerar a seguinte interação:

```
> > > potencia(3, 2)
```

```
9
```

```
> > > potencia(3, -2)
```

```
0.111111111111111111
```

```
> > > potencia
```

```
<function potencia at 0x10f45d0>
```

```
> > > potencia_aux
```

NameError: name 'potencia_aux' is not defined Esta interacção mostra que a função potencia é conhecida pelo Python e que a função potencia aux não é conhecida pelo Python, mesmo depois de este a ter utilizado na execução de potencia(3, 2)!

Na Figura 3.5 apresentamos de um modo informal a estrutura de blocos que definimos para a nova versão da função potencia. Nesta figura, um bloco é indicado como um rectângulo a cheio, dentro do qual aparece a informação do bloco. Note-se que dentro da função potencia existe um bloco que corresponde à função potencia aux. Nesta óptica, o comportamento que obtivemos na interacção anterior está de acordo com a regra associada à estrutura de blocos.

De acordo com esta regra, toda a informação definida dentro de um bloco pertence a esse bloco, e só pode ser usada por esse bloco e pelos blocos definidos dentro dele. Isto faz com que a função potencia aux só possa ser utilizada



3.5. ESTRUTURAC

AO DE FUNC

OES

95

def potencia (x, n):

def potencia aux (b, e):

res = 1

while e > 0:

res = res * x

e = e - 1

return res


```

if n >= 0:
    return potencia aux(x, n)
else:
    return 1 / potencia aux(x, -n)

```

Figura 3.5: Estrutura de blocos da função potencia.

Ambiente global

potencia

função

Figura 3.6: Ambiente global com o nome potencia.

pela função potencia, o que efectivamente evita a possibilidade de utilizarmos directamente a função potencia aux, como a interacção anterior o demonstra.

Para compreender o funcionamento da função potencia, iremos seguir o funcionamento do Python durante a interacção anterior. Ao definir a função potencia cria-se, no ambiente global⁸, a associação entre o nome potencia e uma entidade computacional correspondente a uma função com certos parâmetros e um corpo (Figura 3.6).

Até agora tudo corresponde ao que sabemos, o nome potencia está associado a uma função, cujo corpo é constituído por duas instruções, a primeira é a definição de uma função e a segunda é uma instrução if. Neste âmbito, o Python sabe que potencia é uma função e desconhece o nome potencia aux.

Com a execução de potencia(3, 2), associam-se os parâmetros formais da função potencia aos parâmetros concretos, cria-se um ambiente local e executa-se o corpo da função. Durante esta execução o Python encontra a definição da função potencia aux, criando um nome correspondente no ambiente local como se mostra na Figura 3.7. A segunda instrução do corpo de potencia corresponde a uma instrução if, na qual ambas as alternativas levam à execução de potencia aux, a qual existe no ambiente local. Depois da execução de potencia ⁸Recorde-se a definição de ambiente global apresentada na página 47.



CAPÍTULO 3. FUNC

OES

Ambiente global

potencia

função

Ambiente local a potencia

x

3

n

2

potencia_aux

função

Figura 3.7: Ambientes durante a execução de potencia.

terminar, o ambiente local desaparece, voltando-se à situação apresentada na Figura 3.6, e o Python deixa de conhecer o significado de potencia-aux, o que justifica a interação apresentada.

Uma questão pertinente a levantar neste momento é a de quando definir funções dentro de outras funções e quando definir funções cujos nomes são colocados no ambiente global. Para responder a esta pergunta devemos considerar que a actividade de programação corresponde à construção de componentes computacionais – por exemplo, funções – que podem ser utilizados como caixas pretas por outros componentes. Sob esta perspectiva, durante a actividade de programação a nossa linguagem de programação está a ser sucessivamente enriquecida no sentido em que vão surgindo operações mais potentes, que podem ser usadas como componentes

para desenvolver operações adicionais. Ao escrever um programa devemos decidir se as funções que desenvolvemos devem ser ou não públicas. Por função pública entenda-se uma função que existe no ambiente global e, conseqüentemente, pode ser utilizada por qualquer outra função.

A decisão de definir funções no ambiente global vai prender-se com a utilidade da função e com as restrições impostas à sua utilização. No exemplo do cálculo da potência, é evidente que esta operação é suficientemente importante para ser disponibilizada publicamente; no entanto a operação auxiliar a que esta recorre, potencia aux, é, como discutimos, privada da função potencia e deve estar escondida, evitando utilizações indevidas.

3.5. ESTRUTURAC

```

~
s
AO DE FUNC

```

```

~
s
OES

```

97

No caso do cálculo da raiz quadrada (apresentado na Secção 3.4.5) deve também ser claro que a função calcula raiz não deve ser tornada pública, no sentido em que não queremos que o palpite inicial seja fornecido ao algoritmo; as funções bom palpite e novo palpite são claramente específicas ao modo de calcular a raiz quadrada, e portanto devem ser privadas. De acordo com esta discussão, a função raiz deverá ser definida do seguinte modo (na Figura 3.8 apresentamos a estrutura de blocos da função raiz):

```

def raiz (x):
    def calcula_raiz (x, palpite):
        def bom_palpite (x, palpite):
            return abs(x - palpite * palpite) < delta
        def novo_palpite (x, palpite):
            return (palpite + x / palpite) / 2
        while not bom_palpite(x, palpite):

```

```
palpite = novo_palpite(x, palpite)
```

```
return palpite
```

```
delta = 0.001
```

```
if x >= 0:
```

```
    return calcula_raiz(x, 1)
```

```
else:
```

raise ValueError('raiz: argumento negativo') Com a introdução da estrutura de blocos, podemos classificar os nomes utilizados por uma função em três categorias, nomes locais, livres e globais. Recordemos, da página 72, a definição do procedimento para calcular o quadrado de um número:

```
def quadrado (x):
```

```
    return x * x
```

Sabemos que o nome `x` será associado a um ambiente local durante a avaliação de uma expressão que utilize a função `quadrado`. Esta associação estará activa

98

CAPÍTULO 3. FUNC

~
,

OES

```
def raiz (x):
```

```
    def calcula_raiz (x, palpite):
```

```
        def bom_palpite (x, palpite):
```

```
            return abs(x - palpite * palpite) < delta
```

```
        def novo_palpite (x, palpite):
```

```
            return (palpite + x / palpite) / 2
```

```
        while not bom_palpite(x, palpite):
```

```
            palpite = novo_palpite(x, palpite)
```

```
return palpite
```

```
delta = 0.001
```

```
if x >= 0:
```

```
    return calcula_raiz(x, 1)
```

```
else:
```

```
    print('Erro:
```

```
    raiz, o argumento n~
```

```
    ao pode ser negativo'))
```

Figura 3.8: Estrutura de blocos da função raiz.

durante a avaliação do corpo da função. Numa função, qualquer parâmetro formal corresponde a um nome local, ou seja, apenas tem significado no âmbito do corpo da função.

Nada nos impede de escrever funções que utilizem nomes que não sejam locais, por exemplo, a função

```
def potencia_estranha(n):
```

```
    pot = 1
```

```
    while n > 0:
```

```
        pot = pot * x
```

```
    n = n - 1
```

```
    return pot
```

contém no seu corpo uma referência a um nome (x) que não é um nome local.

Com esta função podemos gerar a seguinte interação:

```
> > > potencia_estranha(3)
```

```
NameError: global name 'x' is not defined
```

```
> > > x = 5
```

```
> > > potencia_estranha(3)
```

125

Na interacção anterior, ao tentarmos avaliar a função potencia estranha, o Python gerou um erro, pois não sabia qual o significado do nome x. A par-

3.5. ESTRUTURAC

```
~  
,
```

AO DE FUNC

```
~  
,
```

OES

99

tir do momento em que dizemos que o nome x está associado ao valor 5, através da instrução x = 5, podemos executar, sem gerar um erro, a função potencia estranha. No entanto esta tem uma particularidade: calcula sempre a potência para a base cujo valor está associado a x (no nosso caso, 5). Um nome, tal como x no nosso exemplo, que apareça no corpo de uma função e que não seja um nome local é chamado um nome não local.

Entre os nomes que não são locais, podemos ainda considerar duas categorias: os nomes globais, aqueles que pertencem ao ambiente global e que são conhecidos por todos os procedimentos (como é o caso de x na interacção anterior), e os que não são globais, a que se dá o nome de livres. Consideremos a seguinte definição que utiliza um nome livre:

```
def potencia_tambem_estranha (x, n):
```

```
def potencia_estranha_2 (n):
```

```
pot = 1
```

```
while n > 0:
```

```
pot = pot * x
```

```
n = n - 1
```

return pot

return potencia_estranha_2(n)

A função potencia estranha 2 utiliza o nome x que não é local (como vimos no exemplo anterior), mas, neste caso, x também não é global, ele é um nome local (na realidade um parâmetro formal) da função potencia_tambem_estranha.

Durante a execução da função potencia_tambem_estranha, o nome x está ligado a um objecto computacional, e a função potencia estranha 2 vai utilizar essa ligação. A utilização de nomes livres é útil quando se desenvolvem funções com muitos parâmetros e que utilizam a estrutura de blocos.

Define-se domínio de um nome como a gama de instruções pelas quais o nome é conhecido, ou seja, o conjunto das instruções onde o nome pode ser utilizado.

Em resumo, os nomes podem ser locais ou não locais, sendo estes últimos ainda

100

CAPÍTULO 3. FUNC

, ~

OES

divididos em livres e globais:

|
|

{ local



nome

global

|

| não local

O domínio dos nomes locais é o corpo da função de que são parâmetros formais ou na qual são definidos; o domínio dos nomes globais é o conjunto de todas as instruções numa interacção em Python. O Python não permite a alteração de nomes livres.

Ao terminar esta secção é importante dizer que a utilização exclusiva de nomes locais permite manter a independência entre funções, no sentido em que toda a comunicação entre elas é limitada à associação dos parâmetros concretos com os parâmetros formais. Quando este tipo de comunicação é mantido, para utilizar uma função, apenas é preciso saber o que ela faz, e não como foi programada.

Já sabemos que isto se chama abstracção procedimental.

Embora a utilização de nomes locais seja vantajosa e deva ser utilizada sempre que possível, é por vezes conveniente permitir o acesso a nomes não locais. O

facto de um nome ser não local significa que o objecto computacional associado a esse nome é partilhado por vários blocos do programa, que o podem consultar e, eventualmente, modificar.

O tipo de domínio utilizado em Python é chamado domínio estático: o domínio de um nome é definido em termos da estrutura do programa (a hierarquia dos seus blocos) e não é influenciado pelo modo como a execução do programa é feita.

3.6

Módulos

Na Secção 3.4 desenvolvemos algumas funções matemáticas elementares, potência, factorial, máximo divisor comum, raiz quadrada e seno. É cada vez mais raro que ao escrever um programa se definam todas as funções que esse programa necessita, é muito mais comum, e produtivo, recorrer aos milhares de linhas de código que outros programadores escreveram e tornaram públicas.

Um módulo (também conhecido por biblioteca) é uma colecção de funções agru-

3.6. MÓDULOS

padas num único ficheiro. As funções existentes no módulo estão relacionadas entre si. Por exemplo, muitas das funções matemáticas de uso comum, como a raiz quadrada e o seno, estão definidas no módulo `math`. Recorrendo à utilização de módulos, quando necessitamos de utilizar uma destas funções, em lugar de a escrevermos a partir do zero, utilizamos as suas definições que já foram programadas por outra pessoa.

Para utilizar um módulo, é necessário importar para o nosso programa as funções definidas nesse módulo. A importação de funções é realizada através da instrução de importação, a qual apresenta a seguinte sintaxe, utilizando a notação BNF:

❖instrução de importação❖ ::= import ❖módulo❖

from ❖módulo❖ import ❖nomes a importar❖

❖módulo❖ ::= ❖nome❖

❖nomes a importar❖ ::= *

❖nomes❖

❖nomes❖ ::= ❖nome❖

❖nome❖, ❖nomes❖

A instrução de importação apresenta duas formas distintas. A primeira destas, `import ❖módulo❖`, indica ao Python para importar para o programa todas as funções existentes no módulo especificado. A partir do momento em que uma instrução de importação é executada, passam a existir no programa nomes correspondentes às funções existentes no módulo, como se de funções embutidas se tratasse.

As funções do módulo são referenciadas através de uma variação de nomes chamada nome composto (ver a Secção 2.3), a qual é definida sintacticamente através da seguinte expressão em notação BNF:

❖nome composto❖ ::= ❖nome simples❖.❖nome simples❖

Neste caso, um ❖nome composto❖ corresponde à especificação do nome do módulo, seguido por um ponto, seguido pelo nome da função. Consideremos o módulo `math`, o qual contém, entre outras, as

funções e os nomes apresentados na Tabela 3.5. Com este módulo, podemos originar a seguinte interação:

```
> > > import math
```

102

CAPÍTULO 3. FUNC

~
,

OES

Python

Matemática Significado

pi

π

Uma aproximação de π

e

e

Uma aproximação de e

$\sin(x)$

$\text{sen}(x)$

O seno de x (x em radianos)

$\cos(x)$

$\cos(x)$

O coseno de x (x em radianos)

$\tan(x)$

$\text{tg}(x)$

A tangente de x (x em radianos)

$\log(x)$

$\ln(x)$

O logaritmo natural de x

$\exp(x)$

e^x

A função inversa de $\ln$

$\text{pow}(x, y)$

x^y

O valor de x levantado a y

$\sqrt{}$

$\text{sqrt}(x)$

x

A raiz quadrada de x

$\text{ceil}(x)$

$\lceil x \rceil$

O maior inteiro superior ou igual a x

$\text{floor}(x)$

$\lfloor x \rfloor$

O maior inteiro inferior ou igual a x

Tabela 3.5: Algumas funções disponíveis no módulo `math`.

```
> > > math.pi
```

```
3.141592653589793
```

```
> > > math.sqrt(4)
```

```
2.0
```

```
> > > math.sin(math.pi/2)
```

1.0

A segunda forma da instrução de importação, `from módulo import nomes a importar`, permite-nos indicar quais as funções ou nomes a importar do módulo, os `nomes a importar`. Após a instrução desta instrução, apenas as funções especificadas são importadas para o nosso programa. Para além disso, os nomes importados não têm que ser referenciados através da indicação de um nome composto, como o mostra a seguinte interação:

```
> > > from math import pi, sin
```

```
> > > pi
```

```
3.141592653589793
```

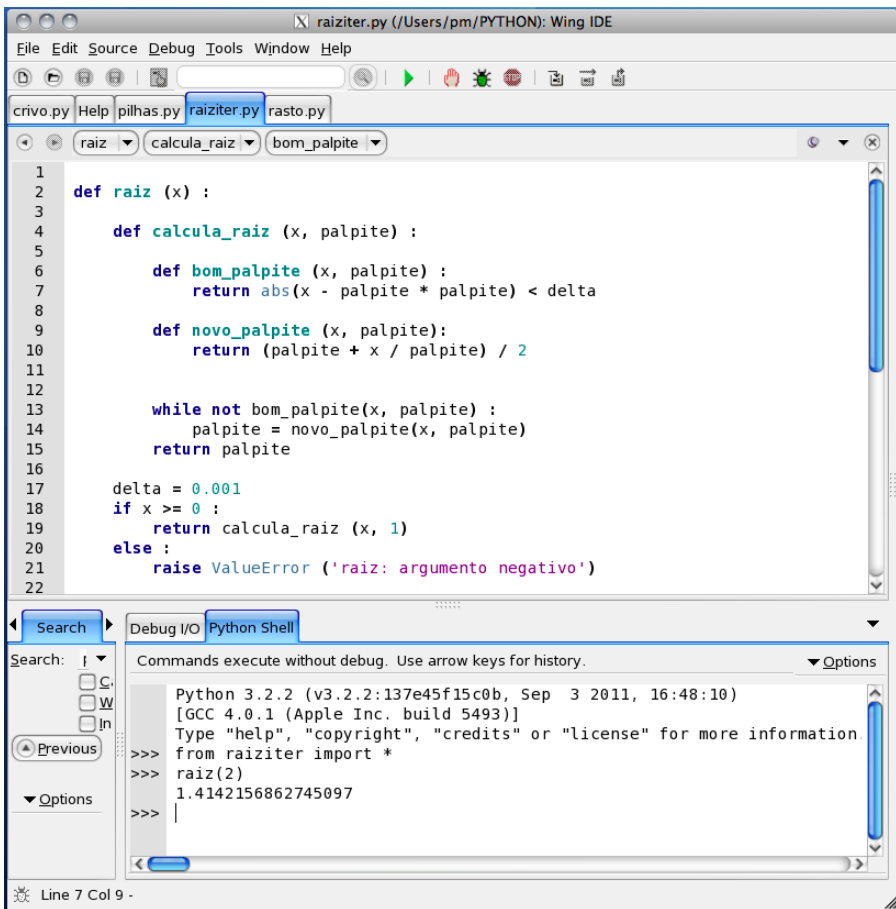
```
> > > sqrt(4)
```

```
NameError: name 'sqrt' is not defined
```

```
> > > sin(pi/2)
```

1.0

Se na especificação de `nomes a importar` utilizarmos o símbolo `*`, então todos os nomes do módulo são importados para o nosso programa, como se ilustra na



3.6. MÓDULOS

103

Figura 3.9: Utilização do módulo raiziter no Wing 101.

interacção:

```
>>> from math import *
```

```
>>> pi
```

```
3.141592653589793
```

```
>>> sqrt(4)
```

```
2.0
```

```
>>> sin(pi/2)
```

Aparentemente, a utilização de `from módulo import *` parece ser preferível a `import módulo`. No entanto, pode acontecer que dois módulos diferentes utilizem o mesmo nome para referirem funções diferentes. Suponhamos que o módulo `m1` define a função `f` e que o módulo `m2` também define a função `f`, mas

104

CAPÍTULO 3. FUNC

~
,

OES

com outro significado. A execução das instruções `from m1 import *`
`from m2 import *`

tem o efeito de “destruir” a definição da função `f` importada do módulo `m1`, substituindo-a pela definição da função `f` importada do módulo `m2` (pois esta importação é feita em segundo lugar). Por outro lado, a execução das instruções `import m1`

`import m2`

permite a coexistência das duas funções `f`, sendo uma delas conhecida por `m1.f` e a outra por `m2.f`.

A lista de todos os módulos disponíveis em Python pode ser consultada em <http://docs.python.org/modindex>. Para a criação de novos módulos por parte do programador basta criar um ficheiro com extensão `.py`, e utilizar um dos comandos de importação de módulos descritos nesta secção, não indicando a extensão do ficheiro. Na Figura 3.9 apresentamos a utilização do módulo `raiziter`, o qual contém a definição da função `raiz quadrada`, tal como apresentada na Secção 3.4.5.

3.7

Notas finais

Apresentámos o modo de definir e utilizar novas funções em Python, e o conceito subjacente à sua utilização, a abstracção procedimental, que consiste em abstrair do modo como as funções realizam as suas tarefas, concentrando-se apenas na tarefa que as funções realizam, ou

seja, a separação do “como” de “o que”.

Discutimos a diferença entre a abordagem da matemática à definição de funções e a abordagem da informática ao mesmo assunto.

Apresentámos exemplos de funções que calculam valores exactos e funções que calculam valores aproximados. Introduzimos o conceito de erro absoluto. O ramo da informática dedicado ao cálculo numérico é conhecido como matemática numérica ou computação numérica. Uma boa abordagem a este tema pode ser consultada em [Ascher and Greif, 2011].

Apresentámos o conceito de estrutura de blocos e a distinção entre nomes globais, livres e locais.

3.8. EXERCÍCIOS

105

Finalmente introduzimos o conceito de módulo.

3.8

Exercícios

1. Escreva uma função com o nome `cinco` que tem o valor `True` se o seu argumento for 5 e `False` no caso contrário. Não pode utilizar uma instrução `if`.
2. Escreva uma função com o nome `bissesto` que determina se um ano é bissesto. Um ano é bissesto se for divisível por 4 e não for divisível por 100, a não ser que seja também divisível por 400. Por exemplo, 1984 é bissesto, 1100 não é, e 2000 é bissesto.
3. Um número primo é um número inteiro maior do que 1 que apenas é divisível por 1 e por si próprio. Por exemplo, 5 é primo porque apenas é divisível por si próprio e por um, ao passo que 6 não é primo pois é divisível por 1, 2, 3, e 6. Os números primos têm um papel muito importante tanto em Matemática como em Informática. Um método simples, mas pouco eficiente, para determinar se um número, n , é primo consiste em testar se n é múltiplo de algum número entre 2 e $\sqrt{n}$. Usando este processo, escreva uma função em Python chamada `primo` que recebe um número inteiro e tem o valor `True` apenas se o seu argumento for primo.
4. Um número n é o n -ésimo primo se for primo e existirem $n - 1$ números primos menores que ele. Usando a função `primo` do exerc

ício anterior, escreva uma função com o nome `n_esimo_primo` que recebe como argumento um número inteiro, `n`, e devolve o `n`-ésimo número primo.

5. Um número inteiro, `n`, diz-se triangular se existir um inteiro `m` tal que $n = 1 + 2 + \dots + (m - 1) + m$. Escreva uma função chamada `triangular` que recebe um número inteiro positivo `n`, e cujo valor é `True` apenas se o número for triangular. No caso de `n` ser 0 deverá devolver `False`. Por exemplo,

```
>>> triangular(6)
```

```
True
```

```
>>> triangular(8)
```

```
False
```

```
106
```

CAPÍTULO 3. FUNC

~
,

OES

6. Escreva uma função em Python que calcula o valor aproximado da série para um determinado valor de `x`:

∞

◆ $x^n = e^x$

$n!$

$n=0$

A sua função não pode utilizar as funções potência nem factorial.

Capítulo 6

Funç

oes revisitadas

‘When I use a word,’ Humpty Dumpty said, in rather a scornful tone, ‘it means just what I choose it to mean neither more nor less.’

‘The question is,’ said Alice, ‘whether you can make words mean so many different things.’

‘The question is,’ said Humpty Dumpty, ‘which is to be the master — that’s all.’

Lewis Carroll, *Through the Looking Glass*

6.1

Funç~

oes recursivas

Nesta secç~ao abordamos um processo de definiç~ao de funç~oes, denominado definiç~ao recursiva ou por recorrência. Diz-se que uma dada entidade é recursiva se ela for definida em termos de si própria. As definiç~oes recursivas s~ao frequentemente utilizadas em matemática. Ao longo deste livro temos utilizado definiç~oes recursivas na apresentaç~ao das express~oes em notaç~ao BNF. Por exemplo, a definiç~ao de $\diamond \text{nomes} \diamond$ apresentada na página 72:

$$\diamond \text{nomes} \diamond ::= \diamond \text{nome} \diamond$$
$$\diamond \text{nome} \diamond, \diamond \text{nomes} \diamond$$

é recursiva pois $\diamond \text{nomes} \diamond$ é definido em termos de si próprio. Esta definiç~ao afirma que $\diamond \text{nomes} \diamond$ é ou um $\diamond \text{nome} \diamond$ ou um $\diamond \text{nome} \diamond$, seguido de uma vírgula, seguida de $\diamond \text{nomes} \diamond$. A utilizaç~ao de recurs~ao é também frequente em imagens, 163



164

CAPÍTULO 6. FUNC

~
s

OES REVISITADAS

Figura 6.1: Exemplo de uma imagem recursiva.

um exemplo das quais apresentamos na Figura 6.11.

Como um exemplo utilizado em matemática de uma definiç~ao recursiva, consideremos a seguinte definiç~ao do conjunto dos números naturais: (1) 1 é um número natural; (2) o sucessor de um número natural é um número natural. A segunda parte desta definiç~ao é recursiva, porque utiliza o conceito de número natural para definir número natural: 2 é um número natural porque é o sucessor do número natural 1 (sabemos que 1 é um número natural pela primeira parte desta definiç~ao).

Outro exemplo típico de uma definiç~ao recursiva utilizada em matemática é a seguinte definiç~ao da func~ao factorial:

se $n = 0$

$n! =$

$n \cdot (n - 1)! \text{ se } n > 0$

Esta definição deve ser lida do seguinte modo: o factorial de 0 ($n = 0$) é 1; se n é maior do que 0, então o factorial de n é dado pelo produto entre n e o factorial de $n - 1$. A definição recursiva da função factorial é mais sugestiva, e rigorosa, do que a seguinte definição que frequentemente é apresentada: $n! = n \cdot (n - 1) \cdot (n - 2) \dots 2 \cdot 1$.

1© United Feature Syndicate, Inc. Nancy and Sluggo are a registered trademark of United Feature Syndicate, Inc.

6.1. FUNC

~
,

OES RECURSIVAS

165

Note-se que na definição não recursiva de factorial, “. . .” significa que existe um padrão que se repete, padrão esse que deve ser descoberto por quem irá calcular o valor da função, ao passo que na definição recursiva, o processo de cálculo dos valores da função está completamente especificado.

O poder das definições recursivas baseia-se na possibilidade de definir um conjunto infinito utilizando uma frase finita. Do mesmo modo, um número arbitrariamente grande de cálculos pode ser especificado através de uma função recursiva, mesmo que esta função não contenha, explicitamente, estruturas de repetição.

Suponhamos que desejávamos definir a função factorial em Python. Podemos usar a definição não recursiva, decodificando o padrão correspondente às re-ticências, dando origem à seguinte função (a qual corresponde a uma variante, utilizando um ciclo for, da função apresentada na Secção 3.4.3): def factorial (n):

```
fact = 1
```

```
for i in range(n, 0, -1):
```

```
fact = fact * i
```

```
return fact
```

ou podemos utilizar directamente a definição recursiva, dando origem à seguinte função:

```
def factorial (n):
```

```
if n == 0:
```

```
return 1
```

```
else:
```

```
return n * factorial(n-1)
```

Esta segunda versão de factorial não contém explicitamente nenhum ciclo.

Como exercício, iremos seguir o funcionamento da versão recursiva da função factorial para calcular o factorial de 3. Ao avaliar factorial(3), o valor 3

é associado ao parâmetro formal n da função factorial, criando-se o ambiente local apresentado na Figura 6.2 (nesta figura e nas seguintes, estamos propositadamente a ignorar as outras ligações que possam existir no ambiente global), no qual o corpo da função é executado. Como a expressão $3 == 0$ tem o valor

166

CAPÍTULO 6. FUNC

, ~

OES REVISITADAS

Ambiente global

...

Ambiente local a factorial

x

3

Figura 6.2: Primeira chamada a factorial.

Ambiente global

...

Ambiente local a factorial

x

3

Ambiente local a factorial

x

2

Figura 6.3: Segunda chamada a factorial.

False, o valor devolvido será $3 * \text{factorial}(2)$.

Encontramos agora uma outra expressão que utiliza a função factorial e esta expressão inicia o cálculo de factorial(2). É importante recordar que neste momento existe um cálculo suspenso, o cálculo de factorial(3). Para calcular factorial(2), o valor 2 é associado ao parâmetro formal n da função factorial, criando-se o ambiente local apresentado na Figura 6.3. Como a expressão $2 == 0$ tem o valor False, o valor devolvido por factorial(2) é $2 *$

factorial(1) e o valor de factorial(1) terá que ser calculado. Dois cálculos da função factorial estão agora suspensos: o cálculo de factorial(3) e o cálculo de factorial(2).

Encontramos novamente outra expressão que utiliza a função factorial e esta expressão inicia o cálculo de factorial(1). Para calcular factorial(1), o valor 1 é associado ao parâmetro formal n da função factorial, criando-se o ambiente local apresentado na Figura 6.4. Como a expressão $1 == 0$ tem o valor False, o valor devolvido por factorial(1) é $1 * \text{factorial}(0)$ e o valor de

6.1. FUNC

~
s

OES RECURSIVAS

Ambiente global

...

Ambiente local a factorial

x

3

Ambiente local a factorial

x

2

Ambiente local a factorial

x

1

Figura 6.4: Terceira chamada a factorial.

`factorial(0)` terá que ser calculado. Três cálculos da função `factorial` estão agora suspensos: o cálculo de `factorial(3)`, `factorial(2)` e `factorial(1)`.

Para calcular `factorial(0)` associa-se o valor 0 ao parâmetro formal `n` da função `factorial`, criando-se o ambiente local apresentado na Figura 6.5 e executa-se o corpo da função. Neste caso a expressão `0 == 0` tem valor `True`, pelo que o valor de `factorial(0)` é 1.

Pode-se agora continuar o cálculo de `factorial(1)`, que é $1 * \text{factorial}(0)$

$= 1 * 1 = 1$, o que, por sua vez, permite calcular `factorial(2)`, cujo valor é $2 * \text{factorial}(1) = 2 * 1 = 2$, o que permite calcular `factorial(3)`, cujo valor é $3 * \text{factorial}(2) = 3 * 2 = 6$.

Em resumo, uma função diz-se recursiva se for definida em termos de si própria.

A ideia fundamental numa função recursiva consiste em definir um problema em termos de uma versão semelhante, embora mais simples, de si próprio. Com efeito, quando definirmos $n!$ como $n \cdot (n - 1)!$ estamos a definir `factorial` em termos de uma versão mais

simples de si próprio, pois o número de que queremos calcular o factorial é mais pequeno. Esta definição é repetida sucessivamente até se atingir uma versão do problema para a qual a solução seja conhecida.

Utiliza-se então essa solução para sucessivamente calcular a solução de cada um dos subproblemas gerados e produzir a resposta desejada.

168

CAPÍTULO 6. FUNC

~
,

OES REVISITADAS

Ambiente global

...

Ambiente local a factorial

x

3

Ambiente local a factorial

x

2

Ambiente local a factorial

x

1

Ambiente local a factorial

x

0

Figura 6.5: Quarta chamada a factorial.

Como vimos, durante a avaliação de uma expressão cujo operador é factorial, gera-se um encadeamento de operações suspensas à espera

do valor de outras operações cujo operador é o próprio factorial. Este encadeamento de operações pode ser representado do seguinte modo:

$3 * \text{factorial}(2)$

$3 (2 \text{ factorial}(1))$

$3 * (2 (1 \text{ factorial}(0)))$

$3 * (2 (1 \ 1))$

$3 (2 \ 1)$

$3 * 2$

6

Podemos constatar que, usando a versão recursiva da função factorial, durante o processo para o cálculo do factorial de um número existem duas fases distintas: (1) numa primeira fase, a execução da operação de multiplicação vai sendo sucessivamente adiada até se atingir o valor que corresponde à parte básica da

6.1. FUNC

~
,

OES RECURSIVAS

169

função factorial (no nosso exemplo, $\text{factorial}(0)$); (2) numa segunda fase, as operações que foram adiadas são executadas.

Tanto as definições recursivas como as funções recursivas são constituídas por duas partes distintas:

1. Uma parte básica, também chamada caso terminal, a qual constitui a versão mais simples do problema para o qual a solução é conhecida. No caso da função factorial esta parte corresponde ao caso em que $n = 0$.

2. Uma parte recursiva, também chamada caso geral, na qual o problema é definido em termos de uma versão mais simples de si próprio. No caso da função factorial, isto corresponde ao caso em que $n > 0$.

Como segundo exemplo, consideremos novamente o algoritmo de Euclides (apresentado na Secção 3.4.4) para o cálculo do máximo divisor comum entre dois números. O algoritmo apresentado para o cálculo do máximo divisor comum afirma que:

1. O máximo divisor comum entre um número e zero é o próprio número.
2. Quando dividimos um número por um menor, o máximo divisor comum entre o resto da divisão e o divisor é o mesmo que o máximo divisor comum entre o dividendo e o divisor.

Este algoritmo corresponde claramente uma definição recursiva. No entanto, na Secção 3.4.4 analisámos o seu comportamento e traduzimo-lo numa função não recursiva. Podemos agora aplicar directamente o algoritmo, dado origem à seguinte função que calcula o máximo divisor comum: $\text{def mdc}(m, n)$:

```
if n == 0:
```

```
    return m
```

```
else:
```

```
    return mdc(n, m % n)
```

Esta função origina o seguinte processo de cálculo, por exemplo, para calcular $\text{mdc}(24, 16)$:

170

CAPÍTULO 6. FUNC

~
s

OES REVISITADAS

$\text{mdc}(24, 16)$

$\text{mdc}(16, 8)$

$\text{mdc}(8, 0)$

8

6.2

Grande parte das linguagens de programação apresentam regras rígidas quanto ao modo como objectos computacionais podem ser manipulados, quanto aos objectos computacionais que podem ser utilizados e quanto ao local onde estes podem ser utilizados. Em Python existe o mínimo possível destas regras. Em Python podemos utilizar um objecto computacional, sempre que a sua utilização fizer sentido. Não faz sentido somar dois valores lógicos ou calcular a conjunção de dois números, mas tanto valores lógicos como números podem ser nomeados, podem ser utilizados como argumentos de uma função ou podem ser o resultado da aplicação de uma função.

De acordo com o cientista inglês Christopher Strachey (1916–1975), os objectos computacionais que podem ser nomeados, que podem ser utilizados como argumentos de funções, que podem ser devolvidos por funções e que podem ser utilizados como componentes de estruturas de informação, são designados por cidadãos de primeira classe.

A ideia subjacente à definição de um objecto computacional como um cidadão de primeira classe é a de que todos estes objectos computacionais têm os mesmos direitos e responsabilidades. De um modo geral as linguagens de programação não são democráticas, dando os direitos de cidadão de primeira classe apenas a alguns dos seus objectos computacionais. O Python, e outras linguagens recentes, é mais democrático, dando este direito a todos, ou quase todos, os objectos computacionais.

Um dos objectos computacionais que tradicionalmente não é tratado como cidadão de primeira classe é a função. Em Python as funções são cidadãos de primeira classe e, consequentemente, podem ser utilizadas como argumentos de outras funções e podem corresponder ao valor devolvido por funções. Nesta secção vamos discutir como esta regalia das funções em Python permite construir abstrações mais poderosas do que as que temos utilizado até aqui.

6.2. FUNC

~
s

6.2.1

Funç~

oes como parâmetros

Nos capítulos anteriores argumentámos que as funç~oes correspondem a abstraç~oes que definem operaç~oes compostas, independentemente dos valores por estas utilizados. Na realidade, ao definirmos a funç~ao quadrado como def quadrado (x):

```
return x * x
```

n~ao estamos a falar do quadrado de um número em particular, mas sim de um padr~ao de cálculo que permite calcular o quadrado de qualquer número. A abstraç~ao procedimental permite introduzir o conceito de quadrado, evitando que tenhamos sempre de exprimi-lo em termos das operaç~oes elementares da linguagem. Sem a introduç~ao do conceito de quadrado, poderíamos calcular o quadrado de qualquer número, mas faltava-nos a abstraç~ao através da qual podemos referir-nos à operaç~ao capturada pelo conceito.

Nesta secç~ao vamos apresentar conceitos mais abstractos do que os que temos utilizado até aqui com a noç~ao de funç~ao. Recordemos mais uma vez que uma funç~ao captura um padr~ao de cálculo. Consideremos agora as seguintes operaç~oes que, embora distintas, englobam um padr~ao de cálculo comum: 1. Soma dos números naturais, inferiores ou iguais a 40: $1 + 2 + \dots + 40$

2. Soma dos quadrados dos números naturais, inferiores ou iguais a 40: $1^2 + 2^2 + \dots + 40^2$

3. Soma dos inversos dos quadrados dos números naturais ímpares, inferiores ou iguais a 40:

1

1

1

+

+

1

2

32

392

Estas operações podem ser realizadas, respectivamente, através das seguintes funções:

```
def soma_inteiros (linf, lsup):
```

172

CAPÍTULO 6. FUNC

~
,

OES REVISITADAS

```
soma = 0
```

```
while linf <= lsup:
```

```
soma = soma + linf
```

```
linf = linf + 1
```

```
return soma
```

```
def soma_quadrados (linf, lsup):
```

```
soma = 0
```

```
while linf <= lsup:
```

```
soma = soma + quadrado(linf)
```

```
linf = linf + 1
```

```
return soma
```

```
def soma_inv_quadrados_impares (linf, lsup): soma = 0
```

```
while linf <= lsup:
```

```
soma = soma + 1 / quadrado(linf)
```

```
linf = linf + 2
```

```
return soma
```

As quais permitem gerar a interação:

```
> > > soma_inteiros(1, 40)
```

```
820
```

```
> > > soma_quadrados(1, 40)
```

```
22140
```

```
> > > soma_inv_quadrados_impares(1, 40) 1.2212031520286797
```

Estas três funções, embora semelhantes, apresentam as seguintes diferenças: (1) o seu nome; (2) o processo de cálculo do termo a ser adicionado; e (3) o processo do cálculo do próximo termo a adicionar.

Independentemente destas diferenças, as três funções anteriores partilham o seguinte padrão de cálculo:

```
def  $\diamond$ nome $\diamond$  (linf, lsup):
```

```
soma = 0
```

6.2. FUNC

~
,

OES DE ORDEM SUPERIOR

```
173
```

```
while linf <= lsup:
```

```
soma = soma +  $\diamond$ calc termo $\diamond$ (linf)
```

```
linf =  $\diamond$ prox $\diamond$ (linf)
```

```
return soma
```

em que $\diamond$ nome $\diamond$, $\diamond$ calc termo $\diamond$ e $\diamond$ prox $\diamond$ são símbolos não terminais que assumem valores particulares para cada uma das funções. A existência deste padrão mostra que existe uma abstração escondida subjacente a este cálculo. Os matemáticos identificaram esta abstração através do conceito de somatório: lsup

$$\blacklozenge f(n) = f(\text{linf}) + f(\text{linf} + 1) + \dots + f(\text{lsup}) \quad n = \text{linf}$$

O poder da abstracção correspondente ao somatório permite lidar com o conceito de soma em vez de tratar apenas com somas particulares. A existência da abstracção correspondente ao somatório leva-nos a pensar em definir uma função correspondente a esta abstracção em vez de apenas utilizar funções que calculam somas particulares.

Um dos processos para tornar as funções mais gerais corresponde a utilizar parâmetros adicionais que indicam quais as operações a efectuar sobre os objectos computacionais manipulados pelas funções, e assim idealizar uma função correspondente a um somatório que, para além de receber a indicação sobre os elementos extremos a somar (linf e lsup), recebe também como parâmetros as funções para calcular um termo do somatório (calc termo) e para calcular o próximo termo a adicionar (prox):

```
def somatorio (calc_termo, linf, prox, lsup): soma = 0
```

```
while linf <= lsup:
```

```
    soma = soma + calc_termo(linf)
```

```
    linf = prox(linf)
```

```
return soma
```

Definindo agora funções para adicionar uma unidade ao seu argumento (inc1), para devolver o seu próprio argumento (identidade), para adicionar duas unidades ao seu argumento (inc2), para calcular o quadrado do seu argumento (quadrado) e para calcular o inverso do quadrado do seu argumento (inv-qua-

174

CAPÍTULO 6. FUNC

~
,

OES REVISITADAS

drado):

```
def inc1 (x):
```

```
    return x + 1
```

```
def identidade (x):
```

```
    return x
```

```
def inc2 (x):
```

```
    return x + 2
```

```
def quadrado (x):
```

```
    return x * x
```

```
def inv_quadrado (x):
```

```
    return 1 / quadrado(x)
```

Obtemos a seguinte interacção:

```
> > > somatorio(identidade, 1, inc1, 40) 820
```

```
> > > somatorio(quadrado, 1, inc1, 40) 22140
```

```
> > > somatorio(inv_quadrado, 1, inc2, 40) 1.2212031520286797
```

Ou seja, utilizando funções como parâmetros, somos capazes de definir a função somatorio que captura o conceito de somatório utilizado em Matemática. Com esta função, cada vez que precisamos de utilizar um somatório, em lugar de escrever um programa, utilizamos um programa existente.

Existem dois aspectos que é importante observar em relação à função anterior. Em primeiro lugar, a função somatorio não foi exactamente utilizada com funções como parâmetros, mas sim com parâmetros que correspondem a nomes de funções. Em segundo lugar, esta filosofia levou-nos a criar funções que podem ter pouco interesse, fora do âmbito das somas que estamos interessados em calcular, por exemplo, a função identidade.

6.2. FUNC

~
,

OES DE ORDEM SUPERIOR

Podemos reparar que o nome da função é, de certo modo, supérfluo, pois o que na realidade nos interessa é saber como calcular o valor da função para um dado argumento

– a função é o conjunto dos pares ordenados. A verdadeira importância do nome da função é a de fornecer um modo de podermos falar sobre ou designar a função. Em 1941, o matemático Alonzo Church inventou uma notação para modelar funções a que se dá o nome de cálculo lambda2. No cálculo lambda, a função que soma 3 ao seu argumento é representada por $\lambda(x)(x + 3)$. Nesta notação, imediatamente a seguir ao símbolo λ aparece a lista dos argumentos da função, a qual é seguida pela expressão designatória que permite calcular o valor da função. Uma das vantagens do cálculo lambda é permitir a utilização de funções sem ter que lhes dar um nome. Para representar a aplicação de uma função a um elemento do seu domínio, escreve-se a função seguida do elemento para o qual se deseja calcular o valor. Assim, $(\lambda(x)(x + 3))(3)$ tem o valor 6; da mesma forma, $(\lambda(x, y)(x \cdot y))(5, 6)$ tem o valor 30.

Em Python existe a possibilidade de definir funções sem nome, as funções anónimas, recorrendo à notação lambda. Uma função anónima é definida através da seguinte expressão em notação BNF:

◆função anónima◆ ::= lambda ◆parâmetros formais◆ :
◆expressão◆

Nesta definição, ◆expressão◆ corresponde a uma expressão em Python, a qual não pode conter ciclos. Este último aspecto não é capturado na nossa definição de ◆função anónima◆.

Ao encontrar uma ◆função anónima◆, o Python cria uma função cujos parâmetros formais correspondem a ◆parâmetros formais◆ e cujo corpo corresponde a ◆expressão◆. Quando esta função anónima é executada, os parâmetros concretos são associados aos parâmetros formais, e o valor da função é o valor da ◆expressão◆.

Note-se que numa função anónima não existe uma instrução return (estando esta implicitamente associada a ◆expressão◆).

Por exemplo, `lambda x : x + 1` é uma função anónima que devolve o valor do seu argumento mais um. Com esta função anónima podemos gerar a interacção:

```
> > > (lambda x : x + 1)(3)
```


CAPÍTULO 6. FUNC

OES REVISITADAS

O que nos interessa fornecer à função somatorio não são os nomes das funções que calculam um termo do somatório e que calculam o próximo termo a adicionar, mas sim as próprias funções. Tendo em atenção, por exemplo, que a função que adiciona 1 ao seu argumento é dado por $\lambda x : x + 1$, independentemente do nome com que é baptizada, podemos utilizar a função somatorio, recorrendo a funções anónimas como mostra a seguinte interacção (a qual pressupõe a definição da função quadrado):

```
> > > somatorio(lambda x : x, 1, lambda x : x + 1, 40) 820
```

```
> > > somatorio(quadrado, 1, lambda x : x + 1, 40) 22140
```

```
> > > somatorio(lambda x : 1/quadrado(x), 1, lambda x : x + 2, 40)
1.2212031520286797
```

Note-se que, a partir da definição da função somatorio, estamos em condições de poder calcular qualquer somatório. Por exemplo, usando a definição da função factorial apresentada na página 165:

```
def factorial (n):
```

```
  if n == 0:
```

```
    return 1
```

```
  else:
```

```
    return n * factorial(n-1)
```

Podemos calcular a soma dos factoriais dos 20 primeiros números naturais através de:

```
> > > somatorio(factorial, 1, lambda x : x + 1, 20)
2561327494111820313
```

Funcionais sobre listas

Com a utilização de listas é vulgar recorrer a um certo número de funções de ordem superior (ou funcionais). Nesta secção apresentamos algumas das funções de ordem superior aplicáveis a listas.

6.2. FUNC

~
,

OES DE ORDEM SUPERIOR

177

• Um transformador é um funcional que recebe como argumentos uma lista e uma operação aplicável aos elementos da lista, e devolve uma lista em que cada elemento resulta da aplicação da operação ao elemento correspondente da lista original. Podemos realizar um transformador através da seguinte função:

def transforma (tr, lista):

res = list()

for e in lista:

res = res + [tr(e)]

return res

A seguinte interacção corresponde a uma utilização de um transformador, utilizando a função quadrado da página 72:

```
> > > transforma(quadrado, [1, 2, 3, 4, 5, 6])
```

```
[1, 4, 9, 16, 25, 36]
```

• Um filtro é um funcional que recebe como argumentos uma lista e um predicado aplicável aos elementos da lista, e devolve a lista constituída apenas pelos elementos da lista original que satisfazem o predicado. Podemos realizar um filtro através da seguinte função: def filtra (teste, lista):

res = list()

for e in lista:

if teste(e):

```
res = res + [e]
```

```
return res
```

A seguinte interação corresponde a uma utilização de um filtro que testa se um número é par:

```
> > > filtra(lambda x : x % 2 == 0, [1, 2, 3, 4, 5, 6])
```

```
[2, 4, 6]
```

- Um acumulador é um funcional que recebe como argumentos uma lista e uma operação aplicável aos elementos da lista, e aplica sucessivamente essa operação aos elementos da lista original, devolvendo o resultado da

178

CAPÍTULO 6. FUNC

~
,

OES REVISITADAS

aplicação da operação a todos os elementos da lista. Podemos realizar um acumulador através da seguinte função:

```
def acumula (fn, lst):
```

```
    res = lst[0]
```

```
    for i in range(1,len(lst)):
```

```
        res = fn(res, lst[i])
```

```
    return res
```

A seguinte interação corresponde a uma utilização de um acumulador, calculando o produto de todos os elementos da lista:

```
> > > acumula(lambda x, y : x * y, [1, 2, 3, 4, 5]) 120
```

Funções como métodos gerais

Nesta secção apresentamos um exemplo que mostra como a utilização de funções como argumentos de funções pode originar métodos gerais de computação, independentemente das funções

envolvidas.

Para isso, apresentamos um processo de cálculo de raízes de equações pelo método do intervalo. Este método é aplicável ao cálculo de raízes de funções contínuas entre dois pontos e baseia-se no corolário do teorema de Bolzano, o qual afirma que, se $f(x)$ é uma função contínua entre os pontos a e b , tais que $f(a) < 0 < f(b)$, então podemos garantir que $f(x)$ tem um zero entre os pontos a e b (Figura 6.6).

Nesta situação, para calcular a raiz da função f no intervalo $[a, b]$, calcula-se o valor de $f(x)$ no ponto médio do intervalo. Se este valor for positivo, podemos garantir que $f(x)$ tem um zero entre a e o valor médio, $(a + b)/2$; se este valor for negativo, podemos garantir que $f(x)$ tem um zero entre o valor médio, $(a + b)/2$, e b . Podemos então repetir o processo com o novo intervalo. Este processo será repetido até que o intervalo em consideração seja suficientemente pequeno.

Consideremos a seguinte função em Python que recebe como argumentos uma função contínua (f) e os extremos de um intervalo no qual a função assume 3Como exercício, o leitor deverá explicar a razão da variável `res` ser inicializada para `lst[0]`

e n

ao para 0.

6.2. FUNC

, ~

OES DE ORDEM SUPERIOR

179

y

f

*

a

*

b

Figura 6.6: Situação à qual o método do intervalo é aplicável.

um valor negativo e um valor positivo, respectivamente, p_{neg} e p_{pos} . Esta função calcula o zero da função no intervalo especificado, utilizando o método do intervalo.

```
def raiz (f, pneg, ppos):
    while not suf_perto(pneg, ppos):
        pmedio = (pneg + ppos) / 2
        if f(pmedio) > 0:
            ppos = pmedio
        elif f(pmedio) < 0:
            pneg = pmedio
        else:
            return pmedio
    return pmedio
```

Esta função parte da existência de uma função para decidir se dois valores estão suficientemente próximos (*suf perto*). Note-se ainda que o *else* da instrução de selecção serve para lidar com os casos em que $f(p_{medio})$ não é nem positivo, nem negativo, ou seja, $f(p_{medio})$ corresponde a um valor nulo e consequentemente é um zero da função.

Para definir quando dois valores estão suficientemente próximos utilizamos a mesma abordagem que usámos no cálculo da raiz quadrada apresentado na Secção 3.4.5: quando a sua diferença em valor absoluto for menor do que um certo limiar (suponhamos que este limiar é 0.001). Podemos assim escrever a

OES REVISITADAS

funç~ao suf perto:

```
def suf_perto (a, b):
```

```
    return abs(a - b) < 0.001
```

Notemos agora que a funç~ao raiz parte do pressuposto que os valores extremos do intervalo correspondem a valores da funç~ao, f , com sinais opostos e que $f(\text{pneg}) < 0 < f(\text{ppos})$. Se algum destes pressupostos n~ao se verificar, a funç~ao n~ao calcula correctamente a raiz. Para evitar este problema potencial, definimos a seguinte funç~ao:

```
def met_intervalo (f, a, b):
```

```
    fa = f(a)
```

```
    fb = f(b)
```

```
    if fa < 0 < fb:
```

```
        return calcula_raiz(f, a, b)
```

```
    elif fb < 0 < fa:
```

```
        return calcula_raiz(f, b, a)
```

```
    else:
```

```
        print('Metodo do intervalo: valores n~
```

```
        ao opostos')
```

Com base nesta funç~ao, podemos obter a seguinte interacç~ao:

```
> > > met_intervalo(lambda x : x x x - 2 * x - 3, 1, 2) 1.8935546875
```

```
> > > met_intervalo(lambda x : x x x - 2 * x - 3, 1, 1.2) Metodo do  
intervalo: valores n~
```

```
ao opostos
```

Sabemos tamb~em que devemos garantir que a funç~ao raiz apenas é utilizada pela funç~ao met_intervalo, o que nos leva a definir a seguinte estrutura de blocos:

```
def met_intervalo (f, a, b):

def calcula_raiz (f, pneg, ppos):

while not suf_perto(pneg, ppos):

pmedio = (pneg + ppos) / 2
```

6.2. FUNC

~
,

OES DE ORDEM SUPERIOR

181

```
if f(pmedio) > 0:

ppos = pmedio

elif f(pmedio) < 0:

pneg = pmedio

else:

return pmedio

return pmedio

def suf_perto (a, b):

return abs(a - b) < 0.001

fa = f(a)

fb = f(b)

if fa < 0 < fb:

return calcula_raiz(f, a, b)

elif fb < 0 < fa:

return calcula_raiz(f, b, a)

else:
```

```
print('Metodo do intervalo: valores n~
```

```
ao opostos')
```

6.2.2

Funç~

oes como valor de funç~

oes

Na secção anterior vimos que em Python as funcções podem ser utilizadas como argumentos de outras funcções. Esta utilização permite a criação de abstrações mais gerais do que as obtidas até agora. Nesta secção, vamos apresentar outro aspecto que torna as funcções cidadãos de primeira classe, discutindo funcções como valores produzidos por funcções.

Cálculo de derivadas

Suponhamos que queremos desenvolver uma função que calcula o valor da derivada, num dado ponto, de uma função real de variável real, f . Esta função real de variável real pode ser, por exemplo, o quadrado de um número, e deverá ser um dos argumentos da função.

Por definição, sendo a um ponto do domínio da função f , a derivada de f no

182

CAPÍTULO 6. FUNC

, ~

OES REVISITADAS

ponto a , representada por $f'(a)$, é dada por: $f'(x) = f(a)$

$$f'(a) = \lim$$

$$x \rightarrow a$$

$$x - a$$

ou, fazendo $h = x - a$,

$$f(a + h) - f(a)$$

$$f'(a) = \lim$$

$$h \rightarrow 0$$

$$h$$

Sendo dx um número suficientemente pequeno, podemos considerar que a seguinte fórmula fornece uma boa aproximação para a derivada da função f no ponto a :

$$f(a + dx) - f(a)$$

$$f'(a) \sim$$

$$=$$

$$dx$$

Podemos agora definir a função derivada ponto, a qual recebe como argumento uma função correspondente a f , e um ponto do seu domínio, a , e produz o valor da derivada nesse ponto:

```
def derivada_ponto (f, a):
```

```
    return (f(a + dx) - f(a)) / dx
```

Esta função, juntamente com a definição do que se entende por algo suficientemente pequeno, dx (por exemplo, 0.00001),

```
dx = 0.00001
```

permite calcular a derivada de funções arbitrárias em pontos particulares do seu domínio. A seguinte interação calcula a derivada da função quadrado (apresentada na página 72) para os pontos 3 e 104:

```
> > > derivada_ponto(quadrado, 3)
```

```
6.000009999951316
```

```
> > > derivada_ponto(quadrado, 10)
```

```
20.00000999942131
```

Em Matemática, após a definição de derivada num ponto, é habitual

definir a função derivada, a qual a cada ponto do domínio da função original associa 4Note-se que a definição da função quadrado não é necessária, pois podemos usar $\lambda x :$

$x * x$. No entanto, a definição

ao da função

ao quadrado torna as nossas expressões

mais

legíveis.

6.2. FUNC

~
,

OES DE ORDEM SUPERIOR

183

o valor da derivada nesse ponto. O conceito de derivada de uma função é suficientemente importante para ser capturado como uma abstração. Repare-se que a derivada num ponto arbitrário x é calculada substituindo x por a na função que apresentámos. O que nos falta na função anterior é capturar o

“conceito de função”. Assim, podemos definir a seguinte função: def derivada (f):

```
def fn_derivada (x):
```

```
    return (f(x + dx) - f(x)) / dx
```

```
return fn_derivada
```

A função anterior recebe como parâmetro uma função, f , e devolve como valor a função (ou seja, $\text{derivada}(f)$ é uma função) que, quando aplicada a um valor, a , produz a derivada da função f para o ponto a . Ou seja, $\text{derivada}(f)(a)$ corresponde à derivada de f no ponto a .

Podemos agora gerar a interação:

```
> > > derivada(quadrado)
```

```
<function fn_derivada at 0x10f45d0>
```

```
> > > derivada(quadrado)(3)
```

```
6.000009999951316
```

```
> > > derivada(quadrado)(10)
```

```
20.00000999942131
```

Nada nos impede de dar um nome a uma função derivada, obtendo a interação:

```
> > > der_quadrado = derivada(quadrado)
```

```
> > > der_quadrado(3)
```

```
6.000009999951316
```

Raízes pelo método de Newton

Na Secção 6.2.1 apresentámos uma solução para o cálculo de raízes de equações pelo método do intervalo. Outro dos métodos muito utilizados para determinar raízes de equações é o método de Newton, o qual é aplicável a funções

184

CAPÍTULO 6. FUNC

, ~

OES REVISITADAS

y *

f

☆☆

☆☆

☆☆

☆(x_n, f(x_n))

☆☆

☆☆

☆

☆

☆

☆

☆

☆

☆

*

☆ x_{n+1}

x_n

x

Figura 6.7: Representação gráfica subjacente ao método de Newton.

diferenciáveis, e consiste em partir de uma aproximação, x_n , para a raiz de uma função diferenciável, f , e calcular, como nova aproximação, o ponto onde a tangente ao gráfico da função no ponto $(x_n, f(x_n))$ intersecta o eixo dos xx (Figura 6.7). É fácil de concluir que a nova aproximação será dada por: $f(x$

x

$n)$

$$n+1 = x_n - f'(x_n)$$

A função matemática que para um dado valor de uma aproximação calcula uma nova aproximação é chamada transformada de Newton. A transformada de Newton é definida por:

$f(x)$

$$t_N(x) = x - f'(x)$$

A determinação da raiz de uma função f , utilizando o método de Newton, corresponde a começar com uma primeira aproximação para a raiz (um palpite), x_0 , e gerar os valores:

x_0

$x_1 = t_N(x_0)$

$x_2 = t_N(x_1)$

...

$x_n = t_N(x_{n-1})$

6.2. FUNC

~
,

OES DE ORDEM SUPERIOR

185

Podemos agora definir as seguintes funções para calcular a raiz de uma função (representada por f), com base num palpite inicial (palpite):

```
def met_newton (f, palpite):
```

```
while not boa_aprox(f(palpite)):
```

```
    palpite = transf_newton(f)(palpite)
```

```
return palpite
```

```
def transf_newton (f):
```

```
    def t_n (x):
```

```
        return x - f(x) / derivada(f)(x)
```

```
    return t_n
```

```
def boa_aprox(x):
```

```
    return abs(x) < dx
```

Com base no método de Newton, e definindo dx como 0.0001, podemos agora calcular os zeros das seguintes funções:

```
>>> met_newton(lambda x : x x x - 2 * x - 3, 1.0)
1.8932892212475259
```

```
>>> from math import *
```

```
> > > met_newton(sin, 2.0)
```

```
3.1415926536589787
```

Suponhamos agora que queríamos definir a função arco de tangente ($\arctg$) a partir da função tangente (tg). Sabemos que $\arctg(x)$ é o número y tal que $x = tg(y)$, ou seja, para um dado x , o valor de $\arctg(x)$ corresponde ao zero da equação $tg(y) - x = 0$. Recorrendo ao método de Newton, podemos então definir:

```
def arctg(x):
```

```
# tg no módulo math tem o nome tan
```

```
return met_newton(lambda y : tan(y) - x, 1.0) obtendo a interação:
```

```
> > > from math import *
```

```
186
```

CAPÍTULO 6. FUNC

~
,

OES REVISITADAS

```
> > > arctg(0.5)
```

```
0.4636478065118169
```

6.3

Programaç~

ao funcional

Existe um paradigma de programação, chamado programação funcional que é baseado exclusivamente na utilização de funções. Um paradigma de programação é um modelo para abordar o modo de raciocinar durante a fase de programação e, conseqüentemente, o modo como os programas são escritos.

O tipo de programação que temos utilizado até agora tem o nome de programação imperativa. Em programação imperativa, um programa é considerado como um conjunto de ordens dadas ao computador, e daí a designação de

“imperativa”, por exemplo, actualiza o valor desta variável, chama esta função, repete a execução destas instruções até que certa condição se verifique.

Uma das operações centrais em programação imperativa corresponde à instrução de atribuição através da qual é criada uma associação entre um nome (considerado como uma variável) e um valor ou é alterado o valor que está associado com um nome (o que corresponde à alteração do valor da variável). A instrução de atribuição leva a considerar variáveis como referências para o local onde o seu valor é guardado e o valor guardado nesse local pode variar. Um outro aspecto essencial em programação imperativa é o conceito de ciclo, uma sequência de instruções associada a uma estrutura que controla o número de vezes que essas instruções são executadas.

Em programação funcional, por outro lado, um programa é considerado como uma função matemática que cumpre os seus objectivos através do cálculo dos valores (ou da avaliação) de outras funções. Em programação funcional não existe o conceito de instrução de atribuição e podem nem existir ciclos. O

conceito de repetição é realizado exclusivamente através da recursão.

Para ilustrar a utilização da programação funcional, consideremos a função potência apresentada na Secção 3.4.2. Uma alternativa para definir x^n é através da seguinte análise de casos:

❖ 1

se $n = 0$

$x^n =$

$x \cdot (x^{n-1})$ se $n > 1$

6.3. PROGRAMAC

~
,

AO FUNCIONAL

187

ou seja, x_0 é 1, e para $n > 0$, x_n é o produto de x pela potência de x

com expoente imediatamente inferior $x \cdot (x^n - 1)$.

Podemos traduzir directamente a esta definição para a seguinte função em Python5:

```
def potencia (x, n):  
  
    if n == 0:  
  
        return 1  
  
    else:  
  
        return x * potencia(x, n-1)
```

Repare-se que estamos perante uma função que corresponde a uma definição recursiva. A função potencia é definida através de uma parte básica, para a qual a solução é conhecida (se $n = 0$, o valor da potência é 1), e de uma parte recursiva, para a qual a potência é definida através de uma versão mais simples de si própria (o expoente é menor).

Nesta função não existe uma definição explícita de repetição, sendo essa repetição tratada implicitamente pela parte recursiva da função. Note-se que também não existe nenhuma instrução de atribuição, os parâmetros formais recebem os seus valores quando a função é chamada (em programação funcional, diz-se que a função é avaliada), sendo os valores devolvidos pela função utilizados nos cálculos subsequentes.

Como segundo exemplo, consideremos a função raiz apresentada na Secção 3.4.5.

A seguinte função corresponde a uma versão funcional de raiz: def raiz (x):

```
def calcula_raiz (x, palpite):  
  
    def bom_palpite (x, palpite):  
  
        return abs(x - palpite * palpite) < 0.001  
  
    def novo_palpite (x, palpite):  
  
        return (palpite + x / palpite) / 2
```

5Propositadamente, não estamos a fazer nenhuma verificação em

relação aos possíveis valores fornecidos a esta função

ao.

188

CAPÍTULO 6. FUNC

~
,

OES REVISITADAS

if bom_palpite(x, palpite):

return palpite

else:

return calcula_raiz(x, novo_palpite(x, palpite)) if x >= 0:

return calcula_raiz(x, 1)

else:

raise ValueError('raiz: argumento negativo') A função calcula raiz apresentada na página 85 foi substituída por uma versão recursiva que evita a utilização do ciclo while.

Como exemplo final, o seguinte programa devolve a soma dos dígitos do número fornecido pelo utilizador, recorrendo à função soma digitos. Repare-se na não existência de instruções de atribuição, sendo o valor fornecido pelo utilizador o parâmetro concreto da função soma digitos.

def prog_soma():

return soma_digitos(eval(input('Escreva um inteiro\n? '))) def soma_digitos(n):

if n == 0:

return n

else:

return n % 10 + soma_digitos(n // 10)

Neste capítulo, introduzimos o conceito de função recursiva, uma função que se utiliza a si própria. Generalizámos o conceito de função através da introdução de dois aspectos, a utilização de funções como argumentos para funções e a utilização de funções que produzem objectos computacionais que correspondem a funções. Esta generalização permite-nos definir abstrações de ordem superior através da construção de funções que correspondem a métodos gerais de cálculo.

6.5. EXERCÍCIOS

189

Introduzimos também um paradigma de programação conhecido por programação funcional.

6.5

Exercícios

6.1 Escreva uma função recursiva em Python que recebe um número inteiro positivo e devolve a soma dos seus dígitos pares. Por exemplo,

```
> > > soma_digitos_pares(234567)
```

12

6.2 Escreva uma função recursiva em Python que recebe um número inteiro positivo e devolve o inteiro correspondente a inverter a ordem dos seus dígitos. Por exemplo,

```
> > > inverte_digitos(7633256)
```

6523367

6.3 Utilizando os funcionais sobre listas escreva uma função que recebe uma lista de inteiros e que devolve a soma dos quadrados dos elementos da lista.

6.4 Defina uma função de ordem superior que recebe funções para calcular as funções reais de variável real f e g e que se comporta como a seguinte função matemática:

$$h(x) = f(x)^2 + 4g(x)^3$$

```
res = 1
valor = a
while valor <= b:
    res = res * fn(valor)
    valor = prox(valor)
return res
```

190

CAPÍTULO 6. FUNC

~5

OES REVISITADAS

Use a função pi para definir a função sin que calcula o seno de um número, usando a seguinte aproximação:



2

◆2

◆2

 $\sin($

X

x

x

$x) = x^1 -$

$1 -$

$1 -$

...

π

2π

3π

A sua função deverá receber o número para o qual se pretende calcular o seno e o número de factores a considerar.

6.6 Escreva a função faz potencia que recebe, como argumento, um inteiro n não negativo e devolve uma função que calcula a potência n de um número.

Por exemplo, faz potencia(3) devolve a função que calcula o cubo do seu argumento. Esta função é ilustrada na seguinte interacção:

```
> > > faz_potencia(3)
```

```
<function f_p at 0x10f4540>
```

```
> > > faz_potencia(3)(2)
```

8

Este exercício é um exemplo da definição de uma função de dois argumentos como uma função de um argumento, cujo valor é uma função de um argumento. Esta técnica, que é útil quando desejamos manter um dos argumentos fixo, enquanto o outro pode variar, é conhecida como método de Curry, e foi concebida por Moses Schönfinkel [Schönfinkel, 1977] e baptizada em honra do matemático americano Haskell B. Curry (1900–1982).

6.7 Considere a função derivada apresentada na página 183. Com base na sua definição escreva uma função que recebe uma função correspondente a uma função e um inteiro n ($n \geq 1$) e devolve a

derivada de ordem n da função. A derivada de ordem n de uma função é a derivada da derivada de ordem $n - 1$.

6.8 Escreva uma função chamada `rasto` que recebe como argumentos uma cadeia de caracteres correspondendo ao nome de uma função, e uma função de um argumento.

A função `rasto` devolve uma função de um argumento que escreve no ecrã a indicação de que a função foi avaliada e o valor do seu argumento, escreve também o resultado da função e tem como valor o resultado da função. Por exemplo, partindo do princípio que a função `quadrado` foi definida, podemos gerar a seguinte interação:

6.5. EXERCÍCIOS

191

$y \ast$

$\ast$

$a = x_0$

x_1

$\dots$

x_n

x

-1

$x_n = b$

Figura 6.8: Área aproximada sob a curva.

```
>>> rasto_quadrado = rasto('quadrado', quadrado)
```

```
>>> rasto_quadrado(3)
```

Avaliação

ao de

quadrado com argumento

Resultado

9

9

6.9 Tendo em atenção que $\sqrt{x}$ é o número y tal que $y^2 = x$, ou seja, para um dado x , o valor de $\sqrt{x}$ corresponde ao zero da equação $y^2 - x = 0$, utilize o método de Newton para escrever uma função que calcula a raiz quadrada de um número.

6.10 Dada uma função f e dois pontos a e b do seu domínio, um dos métodos para calcular a área entre o gráfico da função e o eixo dos xx no intervalo

$[a, b]$ consiste em dividir o intervalo $[a, b]$ em n intervalos de igual tamanho, $[x_0, x_1]$, $[x_1, x_2]$, $\dots$, $[x_{n-2}, x_{n-1}]$, $[x_{n-1}, x_n]$, com $x_0 = a$, $x_n = b$, e $\forall i, j$ $x_i - x_{i-1} = x_j - x_{j-1}$. A área sob o gráfico da função será dada por (Figura 6.8):

n



x

f

$i + x_i - 1$

(

2

$x_i - x_{i-1}$)

$i = 1$

Escreva em Python uma função de ordem superior que recebe como argumento uma função e dois pontos a e b do seu domínio e retorna a área sob o gráfico da função entre a e b .

192

OES REVISITADAS

gumentos uma função (correspondente à função f) e os valores de a e b e que calcula o valor da área entre o gráfico da função e o eixo dos xx no intervalo $[a, b]$. A sua função poderá começar a calcular a área para o intervalo $[x_0, x_1] = [a, b]$ e ir dividindo sucessivamente os intervalos

$[x_i - 1, x_i]$ ao meio, até que o valor da área seja suficientemente bom.

Capítulo 4

Tuplos e ciclos contados

“Then you keep moving round, I suppose?” said Alice.

“Exactly so,” said the Hatter: “as the things get used up.”

“But what happens when you come to the beginning again?” Alice ventured to ask.

Lewis Carroll, *Alice’s Adventures in Wonderland* Até agora, os elementos dos tipos de informação que considerámos correspondem a um único valor, um inteiro, um real ou um valor lógico. Este é o primeiro capítulo em que discutimos tipos estruturados de informação, ou seja, tipos de informação em que os seus elementos estão associados a um agregado de valores.

Recorde-se que um tipo de informação corresponde a um conjunto de entidades, os elementos do tipo, e a um conjunto de operações aplicáveis a essas entidades.

Os tipos de informação cujos elementos estão associados a um agregado de valores são chamados tipos estruturados de informação, tipos de informação não elementares ou estruturas de informação. Sempre que abordamos um tipo estruturado de informação, temos que considerar o modo como os valores estão agregados e as operações que podemos efectuar sobre os elementos do tipo.

4.1

Tuplos

Um tuplo, em Python designado por tuple, é uma sequência de elementos. Os tuplos correspondem à noção matemática de vector.

Em matemática, para nos referirmos aos elementos de um vector, utilizamos índices que caracterizam uni-109

110

CAPÍTULO 4. TUPLOS E CICLOS CONTADOS

15

6

10

12

12

Figura 4.1: Representação gráfica de um tuplo.

vocamente estes elementos. Por exemplo, se x representa um vector com três elementos, estes são caracterizados, respectivamente, por x_1 , x_2 , e x_3 . Analogamente, em Python os elementos de um tuplo são referidos, indicando a posição que o elemento ocupa dentro do tuplo. Tal como em matemática, esta posição tem o nome de índice. Na Figura 4.1 apresentamos, de um modo esquemático, um tuplo com cinco elementos, 15, 6, 10, 12 e 12. O elemento que se encontra na primeira posição do tuplo é 15, o elemento na segunda posição é 6 e assim sucessivamente.

Em Python, a representação externa de um tuplo¹ é definida sintacticamente pelas seguintes expressões em notação BNF2, 3:

$\langle \text{tuplo} \rangle ::= ()$

$(\langle \text{elemento} \rangle, \langle \text{elementos} \rangle)$

$\langle \text{elementos} \rangle ::= \langle \text{nada} \rangle$

$\langle \text{elemento} \rangle$

$\langle \text{elemento} \rangle, \langle \text{elementos} \rangle$

$\langle \text{elemento} \rangle ::= \langle \text{expressão} \rangle$

$\langle \text{tuplo} \rangle$

$\langle \text{lista} \rangle$

❖dicionário❖

❖nada❖ ::=

O tuplo () não tem elementos e é chamado o tuplo vazio. As seguintes entidades representam tuplos em Python (1, 2, 3), (2, True), (1,). Note-se que o último tuplo apenas tem um elemento. A definição sintáctica de um tuplo exige que um tuplo com um elemento contenha esse elemento seguido de uma vírgula, pelo que (1) não corresponde a um tuplo em Python. De acordo 1Recorde-se que a representação externa de uma entidade corresponde ao modo como nós visualizamos essa entidade, independentemente do como esta é representada internamente no computador.

2 É ainda possível representar tuplos sem escrever os parênteses, mas essa alternativa não

ao

é considerada neste livro.

3As definições de ❖lista❖ e ❖dicionário❖ são apresentadas, respectivamente nos capítulos 5 e 8.

4.1. TUPLOS

111

-5

-4

-3

-2

-1

15

6

10

12

12

0
1
2
3
4

Figura 4.2: Valores dos índices de um tuplo.

com as expressões anteriores em notação BNF, $(1, 2,)$ e $(1, 2, 3,)$ são tuplos, respectivamente com 2 e 3 elementos. O tuplo apresentado na Figura 4.1

corresponde a $(15, 6, 10, 12, 12)$.

Note-se que a definição de um tuplo permite que os seus elementos sejam, por sua vez, outros tuplos. Por exemplo, $((1, 2, 3), 4, (5, 6))$ é um tuplo com 3 elementos, sendo o primeiro e o último outros tuplos.

Depois da criação de um tuplo, podemos referir-nos a qualquer dos seus elementos especificando o nome do tuplo e a posição que o elemento desejado ocupa dentro deste. A referência a um elemento de um tuplo corresponde a um nome indexado, o qual é definido através da seguinte expressão em notação BNF:

nome indexado ::= **nome**[**expressão**]

em que **nome** corresponde ao nome do tuplo e **expressão** (que é do tipo inteiro⁴) corresponde à especificação do índice do elemento respectivo. As entidades utilizadas para especificar a posição de um elemento de um tuplo são chamadas índices. Os índices começam no número zero (correspondente ao primeiro elemento do tuplo), aumentando linearmente até ao número de elementos do tuplo menos um; em alternativa, o índice -1 corresponde ao último elemento do tuplo, o índice -2 corresponde ao penúltimo elemento do tuplo e assim sucessivamente, como se mostra na Figura 4.2. Por exemplo, com base no tuplo apresentado na Figura 4.2, podemos gerar a seguinte interação:

```
> > > notas = (15, 6, 10, 12, 12)
```

```
> > > notas
```

```
(15, 6, 10, 12, 12)
```

```
> > > notas[0]
```

4Este aspecto não pode ser especificado utilizando a notação BNF.

112

CAPÍTULO 4. TUPLOS E CICLOS CONTADOS

15

```
> > > notas[-2]
```

12

```
> > > i = 1
```

```
> > > notas[i + 1]
```

10

```
> > > notas[i + 10]
```

IndexError: tuple index out of range

Note-se que na última expressão da interação anterior, tentamos a utilizar o

índice 11 ($= i + 10$), o que origina um erro, pois para o tuplo notas o maior valor do índice é 4.

Consideremos agora a seguinte interação que utiliza um tuplo cujos elementos são outros tuplos:

```
> > > a = ((1, 2, 3), 4, (5, 6))
```

```
> > > a[0]
```

(1, 2, 3)

```
> > > a[0][1]
```

2

A identificação de um elemento de um tuplo (o nome do tuplo seguido do índice dentro de parêntesis rectos) é um nome indexado, pelo que poderemos ser tentados a utilizá-lo como uma variável e, consequentemente, sujeitá-lo a qualquer operação aplicável às variáveis do seu tipo. No entanto, os tuplos em Python são entidades

imutáveis, significando que os elementos de um tuplo não podem ser alterados como o mostra a seguinte interação:

```
>>> a = ((1, 2, 3), 4, (5, 6))
```

```
>>> a[1] = 10
```

TypeError: 'tuple' object does not support item assignment Sobre os tuplos podemos utilizar as funções embutidas apresentadas na Tabela 4.1. Nesta tabela “Universal” significa qualquer tipo. Note-se que as operações $+$ e $*$ são sobrecarregadas, pois também são aplicáveis a inteiros e a reais.

A seguinte interação mostra a utilização de algumas operações sobre tuplos:

4.1. TUPLOS

113

Operação

Tipo dos

Valor

argumentos

$t1 + t2$

Tuplos

A concatenação dos tuplos $t1$ e $t2$.

$t * i$

Tuplo e inteiro

A repetição i vezes do tuplo t .

$t[i1:i2]$

Tuplo e inteiros O sub-tuplo de t entre os índices $i1$ e $i2 - 1$.

e in t

Universal

True se o elemento e pertence ao tuplo t;

e tuplo

False em caso contrário.

tuple()

—

O tuplo vazio.

len(t)

Tuplo

O número de elementos do tuplo t.

Tabela 4.1: Operações embutidas sobre tuplos.

```
>>> a = (1, 2, 3)
```

```
>>> a
```

```
(1, 2, 3)
```

```
>>> b = (7, 8, 9)
```

```
>>> b
```

```
(7, 8, 9)
```

```
>>> a + b
```

```
(1, 2, 3, 7, 8, 9)
```

```
>>> c = a + b
```

```
>>> c[2:4]
```

```
(3, 7)
```

```
>>> a * 3
```

```
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

```
>>> 3 in a
```

True

```
> > > 4 in a
```

False

```
> > > len(a)
```

3

```
> > > a[:2]
```

(1, 2)

```
> > > a[2:]
```

(3,)

```
> > > a[:]
```

(1, 2, 3)

114

CAPÍTULO 4. TUPLOS E CICLOS CONTADOS

As últimas linhas da interação anterior mostram que se na operação `t[e1 : e2]`, um dos índices for omitido, então o Python assume o valor zero se o índice omitido for `e1` ou o maior índice do tuplo se o índice omitido for `e2`.

Consideremos a seguinte interação:

```
> > > a = (3, 4, 5, 6)
```

```
> > > a
```

(3, 4, 5, 6)

```
> > > b = (7, 8)
```

```
> > > b
```

(7, 8)

```
> > > a = a + b
```

```
> > > a
```

(3, 4, 5, 6, 7, 8)

Podemos ser levados a pensar que no penúltimo comando que fornecemos ao Python, `a = a + b`, alterámos o tuplo `a`, o que pode parecer uma violação ao facto de os tuplos serem entidades imutáveis. O que na realidade aconteceu, foi que modificámos o valor da variável `a` que estava associada a um tuplo, sendo esta uma operação perfeitamente válida. Quando afirmámos que os tuplos são imutáveis, queríamos dizer que não podemos alterar um valor de um elemento de um tuplo, podendo perfeitamente criar tuplos a partir de outros tuplos, como a interação anterior o mostra.

Uma das operações que é comum realizar sobre tipos estruturados que correspondem a sequências de elementos, de que os tuplos são um de muitos exemplos, consiste em processar, de um modo idêntico, todos os elementos da sequência.

Como exemplo de uma operação deste tipo, suponhamos que desejávamos escrever uma função que recebe um tuplo e que calcula a soma dos seus elementos.

Esta função deverá inicializar uma variável que conterá o valor da soma para o valor zero e, em seguida, deverá percorrer todos os elementos do tuplo, adicionando o valor de cada um deles à variável que corresponde à soma. Após percorridos todos os elementos do tuplo, a variável correspondente à soma irá conter a soma de todos os elementos. Podemos recorrer a um ciclo `while` para escrever a seguinte função:

4.1. TUPLOS

115

```
def soma_elementos (t):
```

```
    soma = 0
```

```
    i = 0
```

```
    while i < len(t):
```

```
        soma = soma + t[i]
```

```
        i = i + 1
```

```
    return soma
```

Consideremos agora o problema, bastante mais complicado do que o anterior, de escrever uma função, chamada *alisa*, que recebe como argumento um tuplo, cujos elementos podem ser outros tuplos, e que devolve um tuplo contendo todos os elementos correspondentes a tipos elementares de informação (inteiros, reais ou valores lógicos) do tuplo original. Por exemplo, com esta função, pretendemos obter a interação:

```
> > > (1, 2, ((3, ), ((4, ), ), 5), (6, ((7, ), ))) (1, 2, 3, 4, 5, 6, 7)
```

```
> > > alisa((((((5, 6), ), ), ), ), )) (5, 6)
```

Para escrever a função *alisa*, iremos utilizar a função embutida *isinstance*, cuja sintaxe é definida através das seguintes expressões em notação BNF: *isinstance*($\Diamond$ expressão $\Diamond$, $\Diamond$ designação de tipo $\Diamond$)

$\Diamond$ designação de tipo $\Diamond ::= \Diamond$ expressão $\Diamond$

$\Diamond$ tuplo $\Diamond$

A função de tipo lógico *isinstance*, tem o valor *True* apenas se o tipo da expressão que é o seu primeiro argumento corresponde ao seu segundo argumento ou se pertence ao tuplo que é seu segundo argumento. Por exemplo:

```
> > > isinstance(3, int)
```

True

```
> > > isinstance(False, (float, bool)) True
```

```
> > > isinstance((1, 2, 3), tuple)
```

True

```
> > > isinstance(3, float)
```

116

CAPÍTULO 4. TUPLOS E CICLOS CONTADOS

t

i

t[:i]

t[i]

t[i + 1:]

((1, 2), 3, (4, (5)))

0

()

(1, 2)

(3, (4, 5))

(1, 2, 3, (4, 5))

1

(1, 2, 3, (4, 5))

2

(1, 2, 3, (4, 5))

3

(1, 2, 3)

(4, 5)

()

(1, 2, 3, 4, 5)

4

Tabela 4.2: Funcionamento de `alisa(((1, 2), 3, (4, (5))))`.

False

A função `alisa` recebe com argumento um tuplo, `t`, e percorre todos os elementos do tuplo `t`, utilizando um índice, `i`. Ao encontrar um elementos que é um tuplo, a função modifica o tuplo original, gerando um tuplo com todos os elementos antes do índice `i` (`t[:i]`), seguido dos elementos do tuplo correspondente a `t[i]`, seguido de todos os elementos depois do índice `i` (`t[i + 1:]`). Se o elemento não for um tuplo, a função passa a considerar o elemento seguinte, incrementando o valor de `i`. Na Tabela 4.2 apresentamos o

funcionamento desta função para a avaliação de `alisa(((1, 2), 3, (4, (5))))`). Como para certos valores de `i`, a expressão `isinstance(t[i], tuple)` tem o valor `False`, para esses valores não se mostram na Tabela 4.2 os valores de `t[i]`, `t[i]` e `t[i + 1:]`.

```
def alisa (t):
```

```
    i = 0
```

```
    while i < len(t):
```

```
        if isinstance(t[i], tuple):
```

```
            t = t[:i] + t[i] + t[i + 1:]
```

```
        else:
```

```
            i = i + 1
```

```
    return t
```

4.2

Ciclos contados

O ciclo que utilizamos na função soma elementos, apresentada na página 115, obriga-nos a inicializar o índice para o valor zero (`i = 0`) e obriga-nos também a actualizar ao valor do índice após termos somado o valor correspondente (`i`

4.2. CICLOS CONTADOS

117

`= i + 1`). Uma alternativa para o ciclo `while` que utilizamos nessa função é a utilização de um ciclo contado.

Um ciclo contado⁵ é um ciclo cuja execução é controlada por uma variável, designada por variável de controle. Para a variável de controle é especificado o seu valor inicial, a forma de actualizar o valor da variável em cada passagem pelo ciclo e a condição de paragem do ciclo. Um ciclo contado executa repetidamente uma sequência de instruções, para uma sequência de valores da variável de controle.

Em Python, um ciclo contado é realizado através da utilização da instrução `for`, a qual permite especificar a execução repetitiva de

uma instrução composta para uma sequência de valores de uma variável de controle. A sintaxe da instrução for é definida pela seguinte expressão em notação BNF6:

◆instrução for◆ ::= for ◆nome simples◆ in ◆expressão◆: CR

◆instrução composta◆

Na definição sintática da instrução for, ◆expressão◆ representa uma expressão cujo valor corresponde a uma sequência (novamente, este aspecto não pode ser especificado utilizando apenas a notação BNF). Por agora, o único tipo de sequências que encontramos foram os tuplos, embora existam outros tipos de sequências. Nesta definição, a ◆instrução composta◆ corresponde ao corpo do ciclo.

A semântica da instrução for é a seguinte: ao encontrar a instrução for ◆var◆

in ◆expressão◆ : CR ◆inst comp◆, o Python executa as instruções correspondentes a ◆inst comp◆ para os valores da variável ◆var◆ correspondentes aos elementos da sequência resultante da avaliação de ◆expressão◆.

No corpo do ciclo de uma instrução for pode também ser utilizada a instrução break apresentada na página 64. Ao encontrar uma instrução break, o Python termina a execução do ciclo, independentemente do valor da variável que controla o ciclo.

Com a instrução for podemos gerar a seguinte interação:

```
> > > for i in (1, 3, 5):
```

```
...
```

```
print(i)
```

5Em inglês, “counted loop”.

6A palavra “for” traduz-se em português por “para”.

118

CAPÍTULO 4. TUPLOS E CICLOS CONTADOS

...

3

5

Utilizando a instrução `for` podemos agora escrever a seguinte variação da função `soma_elementos`, apresentada na página 115, que recebe um tuplo e devolve a soma de todos os seus elementos:

```
def soma_elementos (t):
```

```
    soma = 0
```

```
    for e in t:
```

```
        soma = soma + e
```

```
    return soma
```

com a qual obtemos a interação:

```
> > > soma_elementos((1, 2))
```

3

O Python fornece também a função embutida `range` que permite a geração de sequências de elementos. A função `range` é definida através das seguintes expressões em notação BNF:

```
range(❖argumentos❖)
```

```
❖argumentos❖ ::= ❖expressão❖
```

```
❖expressão❖, ❖expressão❖
```

```
❖expressão❖, ❖expressão❖, ❖expressão❖
```

Sendo `e1`, `e2` e `e3` expressões cujo valor é um inteiro, a função `range` origina uma progressão aritmética, definida do seguinte modo para cada possibilidade dos seus argumentos:

1. `range(e1)` devolve a sequência contendo os inteiros entre 0 e `e1 - 1`, ou seja devolve o tuplo `(0, 1, . . . , e1 - 1)`. Se `e1 ≤ 0`, devolve o tuplo `()`.

Por exemplo, o valor de `range(10)` corresponde ao tuplo `(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)`.

2. `range(e1, e2)` devolve a sequência contendo os inteiros entre $e1$ e $e2 - 1$, ou seja devolve o tuplo $(e1, e1 + 1, \dots, e2 - 1)$. Se $e2 \leq e1$, devolve o tuplo $()$. Por exemplo, o valor de `range(5, 10)` corresponde ao tuplo $(5, 6, 7, 8, 9)$ e o valor de `range(-3, 3)` corresponde ao tuplo $(-3,$

$-2, -1, 0, 1, 2)$.

3. `range(e1, e2, e3)` devolve a sequência contendo os inteiros que começam em $e1$ e nunca sendo superiores a $e2 - 1$, em que cada elemento da sequência é obtido do anterior somando $e3$, ou seja corresponde ao tuplo $(e1, e1 + e3, e1 + 2 \cdot e3, \dots)$. Novamente, se $e2 \leq e1$, devolve o tuplo $()$. Por exemplo, o valor de `range(2, 20, 3)` corresponde ao tuplo $(2, 5, 8, 11, 14, 17)$.

Recorrendo à função `range` podemos escrever a seguinte função alternativa para calcular a soma dos elementos de um tuplo:

```
def soma_elementos (t):
```

```
    soma = 0
```

```
    for i in range(len(t)):
```

```
        soma = soma + t[i]
```

```
    return soma
```

À primeira vista pode parecer que a utilização de `range` é inútil dado que podemos percorrer todos os elementos de um tuplo `t` usando a instrução `for` e `in t`. Contudo, esta instrução apenas nos permite percorrer os elementos do tuplo, fazendo operações com estes elementos. Suponhamos que desejávamos escrever uma função para determinar se os elementos de um tuplo aparecem ordenados, ou seja, se cada elemento é menor ou igual ao elemento seguinte. A instrução `for` e `in t` embora permita inspeccionar cada elemento do tuplo não nos permite relacioná-lo com o elemento seguinte. Recorrendo à função `range` podemos percorrer o tuplo usando índices, o que já nos permite a comparação desejada como o ilustra a seguinte função:

```
def ordenado (t):
```

```
for i in range(len(t)-1):
```

```
    if t[i] > t[i + 1]:
```

```
        return False
```

```
    return True
```

```
120
```

CAPÍTULO 4. TUPLOS E CICLOS CONTADOS

Note-se que, em contraste com a função soma elementos, a instrução for é executada para os valores de i em $\text{range}(\text{len}(t)-1)$, pois a função ordenado compara cada elemento do tuplo com o seguinte. Se tivesse sido utilizado $\text{range}(\text{len}(t))$, quando i fosse igual a $\text{len}(t)-1$ (o último valor de i neste ciclo), a expressão $t[i] > t[i + 1]$ dava origem a um erro pois $t[\text{len}(t)]$ referencia um índice que não pertence ao tuplo.

Os ciclos while e for, tem características distintas. Assim, põe-se a questão de saber que tipo de ciclo escolher em cada situação.

Em primeiro lugar, notemos que o ciclo while permite fazer tudo o que o ciclo for permite fazer⁷. No entanto, a utilização do ciclo for, quando possível, é mais eficiente do que o ciclo while equivalente. Assim, a regra a seguir na escolha de um ciclo é simples: sempre que possível, utilizar um ciclo for; se tal não for possível, usar um ciclo while.

Convém também notar que existem certas situações em que processamos todos os elementos de um tuplo mas não podemos usar um ciclo for, como acontece com a função alisa apresentada na página 116. Na realidade, nesta função estamos a processar um tuplo que é alterado durante o processamento (o número de elementos do tuplo pode aumentar durante o processamento) e consequentemente não podemos saber à partida quantos elementos vamos considerar.

4.3

Cadeias de caracteres revisitadas

No início do Capítulo 2 introduzimos as cadeias de caracteres como constantes.

Dissemos que uma cadeia de caracteres é qualquer sequência de caracteres delimitados por plicas. Desde então, temos usado cadeias

de caracteres nos nossos programas para produzir mensagens para os utilizadores.

Em Python, as cadeias de caracteres correspondem a um tipo estruturado de informação, designado por `str`, o qual corresponde a uma sequência de caracteres individuais.

As cadeias de caracteres são definidas através das seguintes expressões em notação BNF:

7 Como exercício, deve exprimir o ciclo `for` em termos do ciclo `while`.

8 Da sua designação em inglês, “string”.

4.3. CADEIAS DE CARACTERES REVISITADAS

121

◆cadeia de caracteres◆ ::= '◆carácter◆*'

"◆carácter◆*"

"""◆carácter◆*"""

A definição anterior indica que uma cadeia de caracteres é uma sequência de zero ou mais caracteres delimitados por plicas, por aspas ou por três aspas, de-vendo os símbolos que delimitam a cadeia de caracteres ser iguais (por exemplo

"abc" não é uma cadeia de caracteres). Como condição adicional, não apresentada na definição em notação BNF, os caracteres de uma cadeia de caracteres delimitadas por plicas não podem conter a plica e os caracteres de uma cadeia de caracteres delimitadas por aspas não podem conter aspas. As cadeias de caracteres "" e "" são chamadas cadeias de caracteres vazias. Ao longo do livro utilizamos as plicas para delimitar as cadeias de caracteres.

As cadeias de caracteres delimitadas por três aspas, chamadas cadeias de caracteres de documentação⁹, são usadas para documentar definições. Quando o Python encontra uma cadeia de caracteres de documentação, na linha imediatamente a seguir a uma linha que começa pela palavra `def` (a qual corresponde a uma definição), o Python associa o conteúdo dessa cadeia de caracteres à entidade que está a ser definida. A ideia subjacente é permitir a consulta rápida de informação associada com a entidade definida, recorrendo à função `help`. A função `help(◆nome◆)` mostra no écran a definição

associada a `nome`, bem como o conteúdo da cadeia de caracteres de documentação que lhe está associada.

Por exemplo, suponhamos que em relação à função soma elementos apresentada na página 118, associávamos a seguinte cadeia de caracteres de documentação: `def soma_elementos (t):`

```
"""
```

Recebe um tuplo e devolve a soma dos seus elementos

```
"""
```

```
soma = 0
```

```
for e in t:
```

```
soma = soma + e
```

```
return soma
```

Com esta definição, podemos gerar a seguinte interação: 9Do inglês “docstring”.

122

CAPÍTULO 4. TUPLOS E CICLOS CONTADOS

-5

-4

-3

-2

-1

F

P

1

2

0

1
2
3
4

Figura 4.3: Valores dos índices de uma cadeia de caracteres.

```
> > > help(soma_elementos)
```

Help on function soma_elementos in module __main__:
soma_elementos(t)

Recebe um tuplo e devolve a soma dos seus elementos. Deste modo, podemos rapidamente saber qual a forma de invocação de uma dada função e obter a informação do que a função faz.

Tal como os tuplos, as cadeias de caracteres são tipos imutáveis, no sentido de que não podemos alterar os seus elementos.

Os elementos das cadeias de caracteres são referenciados utilizando um índice, de um modo semelhante ao que é feito em relação aos tuplos. Por exemplo, se `id_fp` for uma variável que corresponde à cadeia de caracteres 'FP 12' (Figura 4.3), então `id_fp[0]` e `id_fp[-1]` correspondem, respectivamente a 'F' e '2'. Cada um destes elementos é uma cadeia de caracteres com apenas um elemento. É

importante notar que '2' não é o mesmo que o inteiro 2, é o carácter "2", como o mostra a seguinte interação:

```
> > > id_fp = 'FP 12'
```

```
> > > id_fp
```

```
'FP 12'
```

```
> > > id_fp[0]
```

```
'F'
```

```
> > > id_fp[-1]
```

```
'2'
```

```
> > > id_fp[-1] == 2
```

False

4.3. CADEIAS DE CARACTERES REVISITADAS

123

Operação

Tipo dos

Valor

argumentos

$s1 + s2$

Cadeias de

A concatenação das cadeias de
caracteres

caracteres $s1$ e $s2$.

$s * i$

Cadeia de

A repetição i vezes da cadeia de
caracteres e inteiro
caracteres s .

$s[i1:i2]$

Cadeia de

A sub-cadeia de caracteres de s
caracteres e inteiros entre os índices $i1$ e $i2 - 1$.
e in s

Cadeias de

True se o elemento e pertence à

caracteres

cadeia de caracteres s; False em

caso contrário.

len(s)

Cadeia de

O número de elementos da cadeia

caracteres

de caracteres s.

Tabela 4.3: Algumas operações embutidas sobre cadeias de caracteres.

Sobre as cadeias de caracteres podemos efectuar as operações indicadas na Tabela 4.3 (note-se que estas operações são semelhantes às apresentadas na Tabela 4.1 e, consequentemente, todas estas operações são sobrecarregadas). A seguinte interacção mostra a utilização de algumas destas operações:

```
> > > cumprimento = 'bom dia!'
```

```
> > > cumprimento[0:3]
```

```
'bom'
```

```
> > > 'ola ' + cumprimento
```

```
'ola bom dia!'
```

```
> > > cumprimento * 3
```

```
'bom dia!bom dia!bom dia!'
```

```
> > > 'z' in cumprimento
```

```
False
```

```
> > > 'ab' in 'abcd'
```

```
True
```

```
> > > len(cumprimento)
```

Como exemplo de utilização das operações existentes sobre cadeias de caracteres, a seguinte função recebe duas cadeias de caracteres e devolve os caracteres da primeira cadeia que também existem na segunda:

124

CAPÍTULO 4. TUPLOS E CICLOS CONTADOS

```
def simbolos_comum (s1, s2):
```

```
# a variável s_comum contém os s´
```

```
simbolos em comum de s1 e s2
```

```
s_comum = ''
```

```
# s_comum é a cadeia de caracteres vazia
```

```
for s in s1:
```

```
if s in s2:
```

```
s_comum = s_comum + s
```

```
return s_comum
```

Com este programa, obtemos a interação:

```
f1 = 'Fundamentos da programaç~
```

```
ao'
```

```
f2 = 'Álgebra linear'
```

```
simbolos_comum(f1, f2)
```

```
'naen a rgraa'
```

Se a cadeia de caracteres s1 contiver caracteres repetidos e estes caracteres existirem em s2, a função simbolos comum apresenta um resultado com caracteres repetidos como o mostra a interação anterior. Podemos modificar a nossa função de modo a que esta não apresente caracteres repetidos do seguinte modo:

```
def simbolos_comum_2 (s1, s2):
```

```
# a variável s_comum contém os s´
```

```
simbolos em comum de s1 e s2
```

```
s_comum = ''
```

```
for s in s1:
```

```
if s in s2 and not s in s_comum:
```

```
s_comum = s_comum + s
```

```
return s_comum
```

A qual permite originar a interacção:

```
>>> f1, f2 = 'Fundamentos de programacão'
```

```
ao', 'Álgebra linear'
```

```
>>> simbolos_comum_2(f1, f2)
```

```
'nae rg'
```

Os caracteres são representados dentro do computador associados a um código numérico. Embora tenham sido concebidos vários códigos para a representação

4.3. CADEIAS DE CARACTERES REVISITADAS

125

de caracteres, os computadores modernos são baseados no código ASCII (American Standard Code for Information Interchange)¹⁰, o qual foi concebido para representar os caracteres da língua inglesa. Por exemplo, as letras maiúsculas são representadas em ASCII pelos inteiros entre 65 a 90. O código ASCII inclui a definição de 128 caracteres, 33 dos quais correspondem a caracteres de controle (muitos dos quais são actualmente obsoletos) e 95 caracteres visíveis, os quais são apresentados na Tabela 4.4. O problema principal associado ao código ASCII corresponde ao facto deste apenas abordar a representação dos caracteres existentes na língua inglesa. Para lidar com a representação de outros caracteres, por exemplo caracteres acentuados, foi desenvolvida uma representação, conhecida por Unicode,¹¹ a qual permite a representação dos caracteres de quase todas as linguagens escritas. Por exemplo, o carácter acentuado “ã” corresponde ao código 227 e o símbolo do

Euro ao código 8364. O código ASCII está contido no Unicode. O Python utiliza o Unicode.

Associado à representação de caracteres, o Python fornece duas funções embutidas, `ord` que recebe um carácter (sob a forma de uma cadeia de caracteres com apenas um elemento) e devolve o código decimal que o representa e a função `chr` que recebe um número inteiro positivo e devolve o carácter (sob a forma de uma cadeia de caracteres com apenas um elemento) representado por esse número. Por exemplo,

```
> > > ord('R')
```

82

```
> > > chr(125)
```

```
'}'
```

O código usado na representação de caracteres introduz uma ordem total entre os caracteres. O Python permite utilizar os operadores relacionais apresentados na Tabela 2.7 para comparar quer caracteres quer cadeias de caracteres. Assim, podemos gerar a seguinte interacção:

```
> > > 'a' < 'b'
```

True

O código ASCII foi publicado pela primeira vez em 1963, tendo sofrido revisões ao longo dos anos, a última das quais em 1986.

As ideias iniciais para o desenvolvimento do Unicode foram lançadas em 1987 por Joe Becker da Xerox e por Lee Collins e Mark Davis da Apple, tendo sido publicados pela primeira vez em 1991.

126

CAPÍTULO 4. TUPLOS E CICLOS CONTADOS

Carácter

Representação

ao

Carácter

Representaç~

ao

Carácter

Representaç~

ao

decimal

decimal

decimal

32

@

64

‘

96

!

33

A

65

a

97

"

34

B

66

b

98

#

35

C

67

c

99

\$

36

D

68

d

100

%

37

E

69

e

101

&

38

F

70

f

102

,

39

G

71

g

103

(

40

H

72

h

104

)

41

I

73

i

105

*

42

J

74

j

106

+

43

K

75

k

107

,

44

L

76

l

108

-

45

M

77

m

109

.

46

N

78

n

110

/

47

O

79

o

111

0

48

P

80

p

112

1

49

Q

81

q

113

2

50

R

82

r

114

3

51

S

83

s

115

4

52

T

84

t

116

5

53

U

85

u

117

6

54

V

86

v

118

7

55

W

87

w

119

8

56

X

88

x

120

9

57

Y

89

y

121

:

58

Z

90

z

122

;

59

[

91

{

123

<

60

\

92

|

124

=

61

]

93

}

125

>

62

^

94

~

Tabela 4.4: Caracteres ASCII visíveis.

4.3. CADEIAS DE CARACTERES REVISITADAS

127

> > > 'a' > 'A'

True

> > > '<' > '<'

False

> > > 'abc' > 'adg'

False

'abc' < 'abcd'

True

Utilizando a representação de caracteres, iremos escrever um programa que recebe uma mensagem (uma cadeia de caracteres) e codifica ou decodifica essa mensagem, utilizando um código de substituição. Uma mensagem é codificada através de um código de substituição, substituindo cada uma das suas letras por outra letra, de acordo com um certo padrão¹². No nosso código, cada letra será substituída pela letra que está quatro posições à sua direita no alfabeto. Assim,

“a” será substituída por “e”, “b” por “f” e assim sucessivamente. As quatro letras do final do alfabeto serão substituídas pelos símbolos do código ASCII situados após o final do alfabeto. Para fazer a codificação, para cada símbolo da nossa mensagem, por exemplo representado pela variável `c`, calculamos o seu código ASCII, dado por `ord(c)`, somamos 4 a esse código e calculamos o símbolo que o substitui, `chr(ord(c) + 4)`. Este símbolo é adicionado ao final da mensagem codificada que estamos a construir.

A seguinte função recebe uma frase por codificar, `f`, e devolve a correspondente mensagem codificada. Para guardar a mensagem codificada, a função usa a variável `codif` que é inicializada com a cadeia de caracteres vazia.

```
def codifica (f):
```

```
    codif = ""
```

```
    for c in f:
```

```
        codif = codif + chr(ord(c)+4)
```

```
    return codif
```

Usando um raciocínio semelhante, podemos escrever a seguinte função que recebe uma mensagem codificada e devolve a mensagem original: def descodifica (f):

120 livro [Sing, 1999] apresenta uma história interessante sobre os vários tipos de métodos para cifrar mensagens.

128

CAPÍTULO 4. TUPLOS E CICLOS CONTADOS

```
desc = ""
```

```
for c in f:
```

```
    desc = desc + chr(ord(c)-4)
```

```
return desc
```

Estamos agora em condições de escrever o nosso programa. Este programa utiliza as funções codifica e descodifica que acabámos de definir. O programa solicita ao utilizador que forneça uma mensagem e qual o tipo de operação a realizar (codificação ou descodificação), após o que efectua a operação solicitada e mostra o resultado obtido.

```
def codificador ():
```

```
    def codifica (f):
```

```
        codif = ""
```

```
        for c in f:
```

```
            codif = codif + chr(ord(c)+4)
```

```
        return codif
```

```
    def descodifica (f):
```



```
desc = ''
```

```
for c in f:
```

```
    desc = desc + chr(ord(c)-4)
```

```
return desc
```

```
original = input('Introduza uma mensagem\n-> ') tipo = input('C  
para codificar ou D para decodificar\n-> ') if tipo == 'C':
```

```
    print('A mensagem codificada é:\n',
```

```
    codifica(original))
```

```
elif tipo == 'D':
```

```
    print('A mensagem decodificada é:\n',
```

```
    decodifica(original))
```

```
else:
```

```
    print('Oops .. n~
```

```
    ao sei o que fazer')
```

4.4. NOTAS FINAIS

129

A seguinte interacção mostra o funcionamento do nosso programa. Repare-se que o programa corresponde a uma função sem argumentos, chamada codificador, a qual não devolve um valor mas sim executa uma sequência de acções.

```
> > > codificador()
```

Introduza uma mensagem

-> Aqui temos um programa bastante simples C para codificar ou D para decodificar

-> C

A mensagem codificada é:

Euym\$xiqsw\$yq\$tvskveqe\$fewxerxi\$wmqtpiw

```
> > > codificador()
```

Introduza uma mensagem

```
-> Euym$xiqsw$yq$tvskveqe$fewxerxi$wmqtpiw C para codificar ou  
D para decodificar
```

```
-> D
```

A mensagem decodificada é:

Aqui temos um programa bastante simples

```
> > > codificador()
```

Introduza uma mensagem

```
-> Euym$xiqsw$yq$tvskveqe$fewxerxi$wmqtpiw C para codificar ou  
D para decodificar
```

```
-> A
```

Oops .. n~

ao sei o que fazer

```
> > >
```

4.4

Notas finais

Este foi o primeiro capítulo em que abordámos o estudo de tipos estruturados de informação. Considerámos dois tipos, os tuplos e as cadeias de caracteres, os quais partilham as propriedades de corresponderem a sequências de elementos e de serem tipos imutáveis. O acesso aos elementos destes tipos é feito recorrendo a um índice correspondendo a um valor inteiro. Introduzimos uma nova

130

CAPÍTULO 4. TUPLOS E CICLOS CONTADOS

instrução de repetição, a instrução for, que corresponde a um ciclo contado.

4.5

Exercícios

1. Escreva em Python a função `duplica` que recebe um tuplo e tem como valor uma tuplo idêntico ao original, mas em que cada elemento está repetido. Por exemplo,

```
> > > duplica((1, 2, 3))
```

```
(1, 1, 2, 2, 3, 3)
```

2. Escreva uma função em Python com o nome `conta_menores` que recebe um tuplo contendo números inteiros e um número inteiro e que devolve o número de elementos do tuplo que são menores do que esse inteiro. Por exemplo,

```
> > > conta_menores((3, 4, 5, 6, 7), 5) 2
```

```
> > > conta_menores((3, 4, 5, 6, 7), 2) 0
```

3. Escreva uma função em Python chamada `maior elemento` que recebe um tuplo contendo números inteiros, e devolve o maior elemento do tuplo.

Por exemplo,

```
> > > maior_elemento((2, 4, 23, 76, 3)) 76
```

4. Defina a função `juntos` que recebe um tuplo contendo inteiros e tem como valor o número de elementos iguais adjacentes. Por exemplo,

```
> > > juntos((1, 2, 2, 3, 4, 4))
```

```
2
```

```
> > > juntos((1, 2, 2, 3, 4))
```

```
1
```

5. Defina uma função, `junta ordenados`, que recebe dois tuplos contendo inteiros, ordenados por ordem crescente. e devolve um tuplo também ordenado com os elementos dos dois tuplos. Por exemplo,

4.5. EXERCÍCIOS

```
> > > junta_ordenados((2, 34, 200, 210), (1, 23)) (1, 2, 23, 34, 200, 210)
```

6. Escreva em Python uma função, chamada soma_els_atomicos, que recebe como argumento um tuplo, cujos elementos podem ser outros tuplos, e que devolve a soma dos elementos correspondentes a tipos elementares de informação que existem no tuplo original. Por exemplo,

```
> > > soma_els_atomicos((3, (((((6, (7, )), ), ), ), ), 2, 1)) 19
```

```
> > > soma_els_atomicos(((((),),),))
```

0

Capítulo 5

Listas

‘Who are you?’ said the Caterpillar.

This was not an encouraging opening for a conversation.

Alice replied, rather shyly, ‘I – I hardly know, Sir, just at present – at least I know who I was when I got up this morning, but I think I must have been changed several times since then.’

Lewis Carroll, Alice’s Adventures in Wonderland Neste capítulo abordamos um novo tipo estruturado de informação, a lista.

Uma lista é uma sequência de elementos de qualquer tipo. Esta definição é semelhante à de um tuplo. Contudo, em oposição aos tuplos, as listas são tipos mutáveis, no sentido de que podemos alterar destrutivamente os seus elementos.

Na maioria das linguagens de programação existem tipos semelhantes às listas existentes em Python, sendo conhecidos por vectores ou por tabelas¹.

5.1

Listas em Python

Uma lista, em Python designada por list, é uma sequência de elementos. Em Python, a representação externa de uma lista é definida sintacticamente pelas seguintes expressões em notação BNF2:

1Em inglês “array”.

2Devemos notar que esta definição não está completa e que a definição de dicionário é apresentada no Capítulo 8.

133

134

CAPÍTULO 5. LISTAS

Operação

Tipo dos

Valor

argumentos

$l_1 + l_2$

Listas

A concatenação das listas l_1 e l_2 .

$l * i$

Lista e inteiro

A repetição i vezes da lista l .

$l[i_1:i_2]$

Lista e inteiros A sublista de l entre os índices i_1 e $i_2 - 1$.

`del(els)`

Lista e

Em que `els` pode ser da forma $l[i]$ ou $l[i_1:i_2]$.

`inteiro(s)`

Remove os elemento(s) especificado(s) da lista l .

`e in l`

Universal

True se o elemento e pertence à lista l;

e lista

False em caso contrário.

list()

—

A lista vazia.

len(l)

Lista

O número de elementos da lista l.

Tabela 5.1: Operações embutidas sobre listas.

❖lista❖ ::= []

[❖elementos❖]

❖elementos❖ ::= ❖elemento❖

❖elemento❖, ❖elementos❖

❖elemento❖ ::= ❖expressão❖

❖tuplo❖

❖lista❖

❖dicionário❖

A lista[] tem o nome de lista vazia. As seguintes entidades representam listas em Python [1, 2, 3], [2, (1, 2)], ['a']. Em oposição aos tuplos, uma lista de um elemento não contém a virgula. Tal como no caso dos tuplos, os elementos de uma lista podem ser, por sua vez, outras listas, pelo que a seguinte entidade é uma lista em Python [1, [2], [[3]]].

Sobre as listas podemos efectuar as operações apresentadas na Tabela 5.1. Note-se a semelhança entre estas operações e as operações sobre tuplos e as operações sobre cadeias de caracteres, apresentadas, respectivamente, nas tabelas 4.1 e 4.3, sendo a excepção a operação

del que existe para listas e que não existe nem para tuplos nem para cadeias de caracteres.

A seguinte interacção mostra a utilização de algumas operações sobre listas:

```
> > > lst1 = [1, 2, 3]
```

```
> > > lst2 = [[4, 5]]
```

```
> > > lst = lst1 + lst2
```

5.1. LISTAS EM PYTHON

135

```
> > > lst
```

```
[1, 2, 3, [4, 5]]
```

```
> > > len(lst)
```

4

```
> > > lst[3]
```

```
[4, 5]
```

```
> > > lst[3][0]
```

4

```
> > > lst[2] = 'a'
```

```
> > > lst
```

```
[1, 2, 'a', [4, 5]]
```

```
> > > del(lst[1])
```

```
> > > lst
```

```
[1, 'a', [4, 5]]
```

```
> > > del(lst[1:])
```

```
> > > lst
```

```
[1]
```

Sendo as listas entidades mutáveis, podemos alterar qualquer dos seus elementos. Na interacção anterior, atribuimos a cadeia de caracteres 'a' ao elemento da lista lst com índice 2 (lst[2] = 'a'), removemos da lista lst o elemento com índice 1 (del(lst[1])), após o que removemos da lista lst todos os elementos com índice igual ou superior a 1 (del(lst[1:])).

Consideremos agora a seguinte interacção:

```
> > > l1 = [1, 2, 3, 4]
```

```
> > > l2 = l1
```

```
> > > l1
```

```
[1, 2, 3, 4]
```

```
> > > l2
```

```
[1, 2, 3, 4]
```

```
> > > l2[1] = 'a'
```

```
> > > l2
```

```
[1, 'a', 3, 4]
```

```
> > > l1
```

```
[1, 'a', 3, 4]
```

136

CAPÍTULO 5. LISTAS

Ambiente global

```
l1
```

```
1
```

```
2
```

```
3
```

```
4
```


Figura 5.1: Ambiente após a definição da lista l1.

Ambiente global

l1

1

2

3

4

l2

Figura 5.2: Ambiente após a atribuição l2 = l1.

Nesta interação, começamos por definir a lista l1, após o que definimos a lista l2 como sendo igual a l1. Ao alterarmos a lista l2 estamos indirectamente a alterar a lista l1. Este comportamento, aparentemente estranho, é explicado quando consideramos o ambiente criado por esta interação. Ao criar a lista l1, o Python dá origem ao ambiente apresentado na Figura 5.1 (usamos uma representação esquemática para listas que é semelhante à usada para os tuplos).

A instrução l2 = l1, define um novo nome, l2, como sendo o valor de l1. Na realidade, a semântica da instrução de atribuição especifica que ao executar a instrução l2 = l1, o Python começa por avaliar a expressão l1 (um nome), sendo o seu valor a entidade associada ao nome (a lista [1, 2, 3, 4]), após o que cria ao nome l2, associando-o ao valor desta expressão. Esta instrução origina o ambiente apresentado na Figura 5.2. As listas l1 e l2 correspondem a pseudónimos³ para a mesma entidade. Assim, ao alterarmos uma delas estamos implicitamente a alterar a outra (Figura 5.3). Esta situação não se verifica com tuplos nem com cadeias de caracteres pois estes são estruturas imutáveis.

Suponhamos agora que continuávamos a interação anterior do seguinte modo:

```
>>> l1 = 10
```

```
>>> l1
```

³Do inglês “alias”.

5.2. MÉTODOS DE PASSAGEM DE PARÂMETROS

137

Ambiente global

l1

1

'a'

3

4

l2

Figura 5.3: Ambiente após a alteração de l1.

Ambiente global

l1

10

1

'a'

3

4

l2

Figura 5.4: Ambiente após nova alteração de l1.

10

> > > l2

[1, 'a', 3, 4]

Esta nova interação dá origem ao ambiente apresentado na Figura 5.4. A variável l1 passa a estar associada ao inteiro 10 e o valor da variável l2 mantém-se inalterado. A segunda parte deste exemplo mostra a diferença entre alterar um elemento de uma lista que é

partilhada por várias variáveis e alterar uma dessas variáveis.

5.2

Métodos de passagem de parâmetros

Com base no exemplo anterior estamos agora em condições de analisar de um modo mais detalhado o processo de comunicação com funções. Como sabemos, quando uma função é avaliada (ou chamada) estabelece-se uma correspondência entre os parâmetros concretos e os parâmetros formais, associando essa que é feita com base na posição que os parâmetros ocupam na lista de parâmetros.

O processo de ligação entre os parâmetros concretos e os parâmetros formais é denominado método de passagem de parâmetros. Existem vários métodos

138

CAPÍTULO 5. LISTAS

de passagem de parâmetros. Cada linguagem de programação utiliza um, ou vários, destes métodos para a comunicação com funções. O Python utiliza dois métodos de passagem de parâmetros, a passagem por valor e a passagem por referência.

5.2.1

Passagem por valor

O método de passagem de parâmetros em Python, quando lida com tipos elementares de informação, é a passagem por valor. Quando um parâmetro é passado por valor, o valor do parâmetro concreto é calculado (independentemente de ser uma constante, uma variável ou uma expressão mais complicada), e esse valor é associado com o parâmetro formal correspondente. Ou seja, utilizando passagem por valor, a função recebe o valor de cada um dos parâmetros e nenhuma informação adicional.

Um parâmetro formal em que seja utilizada a passagem por valor comporta-se, dentro do bloco associado à sua função, como um nome local que é inicializado com o início da avaliação da função.

Para exemplificar, consideremos a função troca definida do seguinte modo: `def troca (x, y):`

```
x, y = y, x
```

```
# os valores s~
```

```
ao trocados
```

e a seguinte interaç~ao que utiliza a funç~ao troca:

```
> > > x = 3
```

```
> > > y = 10
```

```
> > > troca(x, y)
```

```
> > > x
```

```
3
```

```
> > > y
```

```
10
```

As duas primeiras linhas desta interaç~ao têm como efeito a criaç~ao dos nomes x e y no ambiente global como se mostra na Figura 5.5.

Quando o Python invoca a funç~ao troca, avalia os parâmetros concretos e associa os parâmetros concretos aos parâmetros formais da funç~ao troca, criando o ambiente local que se mostra na Figura 5.6.

5.2. MÉTODOS DE PASSAGEM DE PARÂMETROS

139

Ambiente global

x

3

y

10

Figura 5.5: Ambiente global antes da invocaç~ao de troca.

Ambiente global

x

3

y

10

Ambiente local a troca

x

3

y

10

Figura 5.6: Ambiente local criado com a invocação de troca.

A instrução de atribuição $x, y = y, x$ executada pela função troca, altera o ambiente local como se mostra na Figura 5.7. Com efeito, recorde-se da Secção 2.3 que ao encontrar esta instrução, o Python avalia as expressões à direita do símbolo “=”, as quais têm os valores 10 e 3, respectivamente, após o que atribui estas valores às variáveis x e y , respectivamente. Esta instrução tem pois o efeito de trocar os valores das variáveis x e y . Os valores dos nomes locais x e y são alterados, mas isso não vai afectar os nomes x e y que existiam antes da avaliação da função troca.

Quando a função troca termina a sua execução o ambiente que lhe está associado desaparece, voltando-se à situação apresentada na Figura 5.5. Ou seja, quando se utiliza a passagem por valor, a única ligação entre os parâmetros concretos e os parâmetros formais é uma associação unidireccional de valores.

É unidireccional porque é feita do ponto de chamada para a função.

140

CAPÍTULO 5. LISTAS

Ambiente global

x

3

y

3 10

y

10 3

Figura 5.7: Ambientes após a execução da instrução de atribuição.

5.2.2

Passagem por referência

Quando um parâmetro é passado por referência, o que é associado ao parâmetro formal correspondente não é o valor do parâmetro concreto, mas sim a localização na memória do computador que contém o seu valor. Utilizando passagem por referência, os parâmetros formais e os parâmetros concretos vão partilhar o mesmo local (dentro da memória do computador) e, consequentemente, qualquer modificação feita aos parâmetros formais reflecte-se nos parâmetros concretos.

Um parâmetro formal em que seja utilizada a passagem por referência corresponde à mesma variável que o parâmetro concreto correspondente, apenas, eventualmente, com outro nome.

O Python utiliza a passagem por referência sempre que um parâmetro concreto corresponde a uma estrutura de informação. Consideremos agora a função `troca_2` que recebe como argumentos uma lista e dois inteiros e que troca os elementos da lista cujos índices correspondem a esses inteiros.

```
def troca_2 (lst, i1, i2):  
  
    lst[i1], lst[i2] = lst[i2], lst[i1]
```

```
# os valores s~
```

```
ao trocados
```

Com esta função podemos originar a seguinte interacção:

```
> > > lista = [0, 1, 2, 3, 4, 5]  
  
> > > troca_2(lista, 2, 5)  
  
> > > lista
```

5.3. O CRIVO DE ERATÓSTENES

Ambiente global

lista

0

1

2

3

4

5

Ambiente local a troca_2

lst

i1

2

i2

5

Figura 5.8: Ambientes após o início da execução de troca 2.

[0, 1, 5, 3, 4, 2]

Neste caso, o parâmetro concreto `lst` partilha o mesmo espaço de memória que a variável global `lista`, pelo que qualquer alteração a `lst` reflecte-se na variável `lista`. Na Figura 5.8 mostramos os ambientes criados no início da execução da função `troca 2`.

5.3

O Crivo de Eratóstenes

O Crivo de Eratóstenes é um algoritmo para calcular números primos que, segundo a tradição, foi criado pelo matemático grego Eratóstenes (c. 285–194

a.C.), o terceiro bibliotecário chefe da Biblioteca de Alexandria. Para

um dado inteiro positivo n , o algoritmo calcula todos os números primos inferiores a n .

Para isso, começa por criar uma lista com todos os inteiros positivos de 2 a n e selecciona o número 2. Enquanto o número seleccionado for menor que $\sqrt{n}$ executam-se as seguintes acções: (a) removem-se da lista todos os múltiplos desse número; (b) passa-se ao número seguinte na lista. No final do algoritmo, quando o número seleccionado for superior a $\sqrt{n}$, a lista apenas contém números primos.

Vamos agora escrever uma função, crivo que recebe um inteiro positivo n e calcula a lista de números primos inferiores a n de acordo com o Crivo de

142

CAPÍTULO 5. LISTAS

Eratóstenes.

A nossa função deve começar por criar a lista com os inteiros entre 2 e n , após o que percorre a lista para os elementos inferiores a $\sqrt{n}$, removendo da lista todos os múltiplos desse elemento. Partindo do princípio da existência de uma função chamada `remove_multiplos` que recebe uma lista e um número e modifica essa lista, removendo todos os múltiplos desse número, a função crivo será: `def crivo(n):`

```
from math import sqrt
```

```
# Criaç~
```

```
ao da lista com inteiros de 2 a n
```

```
lista = []
```

```
for i in range(2, n + 1):
```

```
    lista = lista + [i]
```

```
# Remoç~
```

```
ao de elementos seleccionados da lista
```

```
i = 0
```

```
while lista[i] < sqrt(n):
```

```
remove_multiplos(lista, lista[i])
```

```
i = i + 1
```

```
return lista
```

É importante perceber a razão de não termos recorrido a um ciclo for na remoção dos elementos da lista. Um ciclo for calcula de antemão o número de vezes que o ciclo será executado, fazendo depois esse número de passagens pelo ciclo com o valor da variável de controlo seguindo uma progressão aritmética.

Este é o ciclo ideal para fazer o processamento de tuplos e de cadeias de caracteres e, tipicamente, é também o ciclo ideal para fazer o processamento de listas.

No entanto, no nosso exemplo, o número de elementos da lista vai diminuindo à medida que o processamento decorre, e daí a necessidade de fazer o controlo da execução do ciclo de um outro modo. Sabemos que temos que processar os elementos da lista inferiores a $\sqrt{n}$ e é este o mecanismo de controle que usamos no ciclo while.

Vamos agora concentrar-nos no desenvolvimento da função remove_multiplos.

Apesar da discussão que acabámos de apresentar em relação à utilização de

5.3. O CRIVO DE ERATÓSTENES

143

uma instrução while, nesta função vamos utilizar um ciclo for que percorre os elementos da lista, do final da lista para o início, deste modo podemos percorrer todos os elementos relevantes da lista, sem termos que nos preocupar com eventuais alterações do índice. Para exemplificar o nosso raciocínio, suponhamos que estamos a percorrer a lista [2, 3, 4, 5, 6, 7, 8, 9, 10], começando no seu último elemento, cujo índice é 8, filtrando os múltiplos de 2. O primeiro elemento considerado é `lst[8]`, cujo valor é 10, pelo que este elemento é removido da lista, o próximo elemento a considerar é `lst[7]`, cujo valor é 9, pelo que, não sendo um múltiplo de 2, este mantém-se na lista, segue-se o elemento `lst[6]`, que é removido da lista e assim sucessivamente. Resta-nos decidir em que elemento da lista devemos parar o nosso processamento. No nosso exemplo, não devemos

processar o elemento `lst[0]`, pois sendo 2, será erradamente retirado da lista. Para isso, usamos uma função, `pos` que recebe a lista e o elemento a ser processado e devolve a posição neste elemento na lista original.

É esta posição que corresponderá ao fim do ciclo.

```
def remove_multiplos(lst, n):
```

```
    for i in range(len(lst)-1, pos(lst, n), -1): if lst[i] % n == 0:
```

```
        del(lst[i])
```

A função `pos` é trivialmente definida do seguinte modo: `def pos (lst, n):`

```
    for i in range(n):
```

```
        if lst[i] > n:
```

```
            return i
```

A função `pos` percorre os elementos da lista, começando na posição 0 até encontrar um elemento maior do o elemento em causa. Como exercício, o leitor deverá compreender a razão de usarmos dentro do ciclo `for` uma instrução `if` com a condição `lst[i] > n` em lugar da condição `lst[i] == n`.

Podemos agora apresentar a função `crivo` de um modo completo, usando a estrutura de blocos:

```
def crivo (n):
```

```
    144
```

CAPÍTULO 5. LISTAS

```
def remove_multiplos(lst, n):
```

```
    def pos (lst, n):
```

```
        for i in range(n):
```

```
            if lst[i] > n:
```

```
                return i
```

```
for i in range(len(lst)-1, pos(lst, n), -1): if lst[i] % n == 0:
```

```
del(lst[i])
```

```
from math import sqrt
```

```
lista = []
```

```
for i in range(2, n + 1):
```

```
lista = lista + [i]
```

```
i = 0
```

```
while lista[i] < sqrt(n):
```

```
remove_multiplos(lista, lista[i])
```

```
i = i + 1
```

```
return lista
```

Com esta função geramos a interacção:

```
> > > crivo(60)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59]
```

5.4

Algoritmos de procura

A procura de informação é uma das nossas actividades quotidianas. Procuramos palavras no Google, palavras em dicionários, livros em bibliotecas, produtos em supermercados, *etc.* Na função pos (apresentada na página 143) procurámos a posição de um número numa lista. Com o aumento constante da quantidade de informação armazenada por computadores, é natural que a procura de informação se tenha tornado uma das actividades preponderantes num sistema

5.4. ALGORITMOS DE PROCURA

145

computacional. Embora a localização de um dado elemento entre um conjunto de elementos pareça ser muito simples, o desenvolvimento

de programas eficientes de procura levanta muitos problemas. Como exemplo, suponhamos que procurávamos a palavra “Python” no Google, a nossa procura devolve cerca de 198 milhões de resultados, tendo a procura demorado 0.26 segundos. Imagine-se quanto demoraria esta procura se o Google não utilizasse métodos de procura altamente sofisticados.

A procura de informação é frequentemente executada recorrendo a listas. Nesta secção, apresentamos dois métodos para procurar um elemento numa lista, a procura sequencial e a procura binária. Para estes dois tipos de procura, iremos desenvolver uma função com o nome procura que recebe como argumentos uma lista (lst) e um elemento (a que chamamos chave). Esta função devolve um inteiro positivo correspondente ao índice do elemento da lista cujo valor é igual a chave (estamos a partir do pressuposto que a lista lst não tem elementos repetidos). Vamos também convencionar que se a lista não contiver nenhum elemento igual a chave, a função devolve -1.

5.4.1

Procura sequencial

Uma das maneiras mais simples de procurar um dado elemento numa lista consiste em começar no primeiro elemento da lista e comparar sucessivamente o elemento procurado com o elemento na lista. Este processo é repetido até que o elemento seja encontrado ou o fim da lista seja atingido. Este método de procura é chamado procura sequencial, ou procura linear. Apresentamos a função procura utilizando a procura sequencial:

```
def procura (lst, chave):
```

```
    for i in range(len(lst)):
```

```
        if lst[i] == chave:
```

```
            return i
```

```
    return -1
```

Com a qual podemos obter a interacção:

```
> > > lst = [2, 3, 1, -4, 12, 9]
```

```
> > > procura(lst, 1)
```

CAPÍTULO 5. LISTAS

2

> > > procura(lst, 5)

-1

5.4.2

Procura binária

A procura sequencial é muito simples, mas pode exigir a inspecção de todos os elementos da lista, o que acontece sempre que o elemento que estamos a procurar não se encontra na lista.

A procura binária é uma alternativa, mais eficiente, à procura sequencial, exigindo, contudo, que os elementos sobre os quais a procura está a ser efectuada se encontrem ordenados. Utilizando a procura binária, consideramos em primeiro lugar o elemento que se encontra no meio da lista: 1. Se este elemento é maior do que o elemento que estamos a procurar então podemos garantir que o elemento que estamos a procurar não se encontra na segunda metade da lista. Repetimos então o processo da procura binária para a primeira metade da lista.

2. Se o elemento no meio da lista é menor do que o elemento que estamos a procurar então podemos garantir que o elemento que estamos a procurar não se encontra na primeira metade da lista. Repetimos então o processo da procura binária para a segunda metade da lista.

3. Se o elemento no meio da lista for igual ao elemento que estamos a procurar, então a procura termina.

Note-se que, em cada passo, a procura binária reduz o número de elementos a considerar para metade (e daí o seu nome).

A seguinte função utiliza a procura binária. Esta função pressupõe que a lista `lst` está ordenada. As variáveis `linf` e `lsup` representam, respectivamente, o menor e o maior índice da gama dos elementos que estamos a considerar.

`def procura (lst, chave):`

```
linf = 0
```

```
lsup = len(lst) - 1
```

5.5. ALGORITMOS DE ORDENAC

~
,

AO

147

```
while linf <= lsup:
```

```
    meio = (linf + lsup) // 2
```

```
    if chave == lst[meio]:
```

```
        return meio
```

```
    elif chave > lst[meio]:
```

```
        linf = meio + 1
```

```
    else:
```

```
        lsup = meio - 1
```

```
return -1
```

A procura binária exige que mantenhamos uma lista ordenada contendo os elementos em que queremos efectuar a procura, consequentemente, o custo computacional da inserção de um novo elemento na lista é maior se usarmos este método do que se usarmos a procura sequencial. Contudo, como as procuras são normalmente mais frequentes do que as inserções, é preferível utilizar uma lista ordenada.

5.5

Algoritmos de ordenaç

ao

Um conjunto de elementos é normalmente ordenado para facilitar a procura.

Na nossa vida quotidiana, ordenamos as coisas para tornar a procura

mais fácil, e não porque somos arrumados. Em programação, a utilização de listas ordenadas permite a escrita de algoritmos de procura mais eficientes, por exemplo, a procura binária em lugar da procura sequencial. Nesta secção, abordamos o estudo de alguns algoritmos para ordenar os valores contidos numa lista.

Os algoritmos de ordenação podem ser divididos em dois grandes grupos, os algoritmos de ordenação interna e os algoritmos de ordenação externa. Os algoritmos de ordenação interna ordenam um conjunto de elementos que estão simultaneamente armazenados em memória, por exemplo, numa lista. Os algoritmos de ordenação externa ordenam elementos que, devido à sua quantidade, não podem estar simultaneamente em memória, estando parte deles armazenados algures no computador (num ficheiro, para ser mais preciso). Neste segundo caso, em cada momento apenas estão em memória parte dos elementos a ordenar. Neste livro apenas consideramos algoritmos de ordenação interna.

Para apresentar os algoritmos de ordenação, vamos desenvolver várias versões

148

CAPÍTULO 5. LISTAS

de uma função chamada ordena. Esta função recebe como parâmetro uma lista contendo os elementos a ordenar (correspondendo ao parâmetro formal lst), a qual, após execução da função, contém os elementos ordenados. Cada versão que apresentamos desta função corresponde a um algoritmo de ordenação.

5.5.1

Ordenação

ao por borbulhamento

O primeiro algoritmo de ordenação que vamos considerar é conhecido por ordenação ao por borbulhamento 4. A ideia básica subjacente a este algoritmo consiste em percorrer os elementos a ordenar, comparando elementos adjacentes, trocando os pares de elementos que se encontram fora de ordem, ou seja, que não estão ordenados. De um modo geral, uma única passagem pela sequência de elementos não ordena a lista, pelo que é necessário efectuar várias passagens.

A lista encontra-se ordenada quando se efectua uma passagem completa em que não é necessário trocar a ordem de nenhum elemento. A razão por que este método é conhecido por “ordenação por borbulhamento” provém do facto de os menores elementos da lista se movimentarem no sentido do início da lista, como bolhas que se libertam dentro de um recipiente com um líquido.

A seguinte função utiliza a ordenação por borbulhamento. Esta função utiliza a variável de tipo lógico, nenhuma troca, cujo valor é False se, durante uma passagem pela lista, se efectua alguma troca de elementos, e tem o valor True em caso contrário. Esta variável é inicializada para False na segunda linha da função de modo a que o ciclo while seja executado.

Dado que em cada passagem pelo ciclo colocamos o maior elemento da lista na sua posição correcta⁵, cada passagem subsequente pelos elementos da lista considera um valor a menos e daí a razão da variável maior índice, que contém o maior índice até ao qual a ordenação se processa.

```
def ordena (lst):
```

```
    maior_indice = len(lst) - 1
```

```
    nenhuma_troca = False
```

```
    while not nenhuma_troca:
```

```
        4Em inglês, “bubble sort”.
```

5Como exercício, o leitor deve convencer-se deste facto.

5.5. ALGORITMOS DE ORDENAC

~
,

AO

149

```
    nenhuma_troca = True
```

```
    for i in range(maior_indice):
```

```
        if lst[i] > lst[i + 1]:
```

```
            lst[i], lst[i + 1] = lst[i + 1], lst[i]
```

```
nenhuma_troca = False
```

```
maior_indice = maior_indice - 1
```

5.5.2

Ordenaç~

ao Shell

Uma variante da ordenaç~ao por borbulhamento, a ordenaç~ao Shell 6 consiste em comparar e trocar, n~ao os elementos adjacentes, mas sim os elementos separados por um certo intervalo. Após uma ordenaç~ao completa, do tipo borbulhamento, com um certo intervalo, esse intervalo é dividido ao meio e o processo repete-se com o novo intervalo. Este processo é repetido até que o intervalo seja 1

(correspondendo a uma ordenaç~ao por borbulhamento). Como intervalo inicial, considera-se metade do número de elementos a ordenar. A ordenaç~ao Shell é mais eficiente do que a ordenaç~ao por borbulhamento, porque as primeiras passagens que consideram apenas um subconjunto dos elementos a ordenar permitem uma arrumaç~ao grosseira dos elementos da lista e as últimas passagens, que consideram todos os elementos, já os encontram parcialmente ordenados. A seguinte funç~ao corresponde à ordenaç~ao Shell: def ordena (lst):

```
intervalo = len(lst) // 2
```

```
while not intervalo == 0:
```

```
    nenhuma_troca = False
```

```
    while not nenhuma_troca:
```

```
        nenhuma_troca = True
```

```
        for i in range(len(lst)-intervalo):
```

```
            if lst[i] > lst[i + intervalo]:
```

6Em inglês, “Shell sort”, em honra ao seu criador Donald Shell [Shell, 1959].

```
lst[i], lst[i + intervalo] = \
```

```
lst[i + intervalo], lst[i]
```

```
nenhuma_troca = False
```

```
intervalo = intervalo // 2
```

5.5.3

Ordenaç~

ao por selecç~

ao

Uma terceira alternativa de ordenaç~ao que iremos considerar, a ordenaç~ao por selecç~ao⁷, consiste em percorrer os elementos a ordenar e, em cada passagem, colocar um elemento na sua posiç~ao correcta. Na primeira passagem, coloca-se o menor elemento na sua posiç~ao correcta, na segunda passagem, o segundo menor, e assim sucessivamente. A seguinte funç~ao efectua a ordenaç~ao por selecç~ao:

```
def ordena (lst):
```

```
    for i in range(len(lst)):
```

```
        pos_menor = i
```

```
        for j in range(i + 1, len(lst)):
```

```
            if lst[j] < lst[pos_menor]:
```

```
                pos_menor = j
```

```
        lst[i], lst[pos_menor] = lst[pos_menor], lst[i]
```

5.6

Exemplo

Para ilustrar a utilizaç~ao de algoritmos de procura e de ordenaç~ao, vamos desenvolver um programa que utiliza duas listas, nomes e telefones, contendo, respectivamente nomes de pessoas e números de telefones. Assumimos que o número de telefone de uma pessoa está armazenado na lista telefones na mesma posiç~ao que o nome dessa

pessoa está armazenado na lista `nomes`. Por exemplo, o número de telefone da pessoa cujo nome é `nomes[3]`, está armazenado em `telefones[3]`. Listas com esta propriedade são chamadas listas 7Em inglês, “selection sort”.

5.6. EXEMPLO

151

paralelas. Assumimos também que cada pessoa tem, no máximo, um número de telefone.

O programa interacciona com um utilizador, ao qual fornece o número de telefone da pessoa cujo nome é fornecido ao programa. Esta interacção é repetida até que o utilizador forneça, como nome, a palavra fim.

O nosso programa utiliza uma abordagem muito simplista, na qual as listas são definidas dentro do próprio programa. Uma abordagem mais realista poderia ser obtida através da leitura desta informação do exterior, o que requer a utilização de ficheiros, os quais são apresentados no Capítulo 9.

O programa utiliza a função procura correspondente à procura binária, a qual foi apresentada na secção 5.4.2, e que não é repetida no nosso programa. O algoritmo para ordenar as listas de nomes e de telefones corresponde à ordenação por selecção apresentada na secção 5.5.3. Contudo, como estamos a lidar com listas paralelas, o algoritmo que usamos corresponde a uma variante da ordenação por selecção. Na realidade, a função ordena nomes utiliza a ordenação por selecção para ordenar os nomes das pessoas, a lista que é usada para procurar os nomes, mas sempre que dois nomes são trocados, a mesma acção é aplicada aos respectivos números de telefone.

```
def lista_tel ():
```

```
def ordena_nomes (pessoas, telef):
```

```
for i in range(len(pessoas)):
```

```
    pos_menor = i
```

```
    for j in range(i + 1, len(pessoas)):
```

```
        if pessoas[j] < pessoas[pos_menor]:
```

```

pos_menor = j

pessoas[i], pessoas[pos_menor] = \
pessoas[pos_menor], pessoas[i]

telef[i], telef[pos_menor] = \
telef[pos_menor], telef[i]

pessoas = ['Ricardo Saldanha', 'Francisco Nobre', \
'Leonor Martins', 'Hugo Dias', 'Luiz Leite', \
'Ana Pacheco', 'Fausto Almeida']
152

```

CAPÍTULO 5. LISTAS

```

telefones = [211234567, 919876543, 937659862, 964876347, \
218769800, 914365986, 229866450]

ordena_nomes(pessoas, telefones)

quem = input('Qual o nome?\n(fim para terminar)\n? ') while quem != 'fim':

pos_tel = procura(pessoas, quem)

if pos_tel == -1:

print('Telefone desconhecido')

else:

print('O telefone é: ', telefones[pos_tel])

quem = input('Qual o nome?\n(fim para terminar)\n? ')
5.7

Consideraç~
oes sobre eficiência

```

Dissemos que a procura binária é mais eficiente do que a procura sequencial, e que a ordenação Shell é mais eficiente do que a ordenação por borbulhamento.

Nesta secção, discutimos métodos para comparar a eficiência de algoritmos. O

estudo da eficiência de um algoritmo é um aspecto importante em informática, porque fornece uma medida grosseira do trabalho envolvido na execução de um algoritmo.

Um dos processos para determinar o trabalho envolvido na execução de um algoritmo consiste em considerar um conjunto de dados e seguir a execução do algoritmo para esses dados. Este aspecto está ilustrado nas figuras 5.9 a 5.11, em que mostramos a evolução da posição relativa dos elementos da lista [4, 8, 17, 3, 11, 2], utilizando os três métodos de ordenação que apresentámos.

Nestas figuras, cada linha corresponde a uma passagem pela lista, um arco ligando duas posições da lista significa que os elementos da lista nessas posições foram comparados e uma seta por baixo de duas posições da lista significa que os elementos nessas posições (os quais estão indicados nas extremidades da seta) foram trocados durante uma passagem.

Esta abordagem, contudo, tem a desvantagem do trabalho envolvido na execução de um algoritmo poder variar drasticamente com os dados que lhe são fornecidos (por exemplo, se os dados estiverem ordenados, a ordenação por

5.7. CONSIDERAÇÕES

~
,

OES SOBRE EFICIÊNCIA

153

Primeira passagem

4

8

17

3

11

2

17

3

17

2

17

11

Segunda passagem

4

8

3

11

2

17

8

3

2

11

Terceira passagem

4

3

8

2

11

17

4

3

8

2

Quarta passagem

3

4

2

8

11

17

4

2

Quinta passagem

3

2

4

8

11

17

3

2

Figura 5.9: Padrão de ordenação utilizando a ordenação por borbulhamento.

Primeira passagem

4

8

17

3

11

2

(intervalo = 3)

4

3

17

2

Segunda passagem

3

8

2

4

11

17

(intervalo = 3)

Primeira passagem

3

8

2

4

11

17

(intervalo = 1)

8

2

8

4

Segunda passagem

3

2

4

8

11

17

(intervalo = 1)

3

2

Terceira passagem

2

3

4

8

11

17

(intervalo = 1)

Figura 5.10: Padrão de ordenação utilizando a ordenação Shell.

154

CAPÍTULO 5. LISTAS

Primeira passagem

4

8

17

3

11

2

4

2

Segunda passagem

2

8

17

3

11

4

8

3

Terceira passagem

2

3

17

8

11

4

17

4

Quarta passagem

2

3

4

8

11

17

Quinta passagem

2

3

4

8

11

17

Figura 5.11: Padrão de ordenação utilizando a ordenação por selecção.

borbulhamento só necessita de uma passagem). Por esta razão, é importante encontrar uma medida do trabalho envolvido na execução de um algoritmo que seja independente dos valores particulares dos

dados que são utilizados na sua execução.

Este problema é resolvido com outro modo de descrever o trabalho envolvido na execução de um algoritmo, que recorre à noção de ordem de crescimento⁸ para obter uma avaliação grosseira dos recursos exigidos pelo algoritmo à medida que a quantidade de dados manipulados aumenta. A ideia subjacente a esta abordagem é isolar uma operação que seja fundamental para o algoritmo, e contar o número de vezes que esta operação é executada. Para o caso dos algoritmos de procura e de ordenação, esta operação é a comparação (dados dois elementos, qual deles é o maior?). Segundo este processo, a medida de eficiência de um algoritmo de procura ou de ordenação é o número de comparações que estes efectuam. Dados dois algoritmos, diremos que o algoritmo que efectuar o menor número de comparações é o mais eficiente.

Na procura sequencial, percorremos a sequência de elementos até encontrar o elemento desejado. Se o elemento desejado se encontrar na primeira posição da ⁸Do inglês, “order of growth”.

5.7. CONSIDERAÇÕES

~
,

OES SOBRE EFICIÊNCIA

155

lista, necessitamos apenas de uma comparação mas poderemos ter de percorrer toda a lista, se o elemento procurado não se encontrar na lista. O número médio de comparações cai entre estes dois extremos, pelo que o número médio de comparações na procura sequencial é:

n

2

em que n é o número de elementos na lista.

Na procura binária somos capazes de reduzir para metade o número de elementos a considerar sempre que efectuamos uma comparação (ver a função apresentada na página 146). Assim, se começarmos com n elementos, o número de elementos depois de uma passagem é $n/2$; o número de elementos depois de duas passagens é $n/4$ e assim sucessivamente. No caso geral, o número de elementos depois de i passagens é $n/2^i$. O algoritmo termina quando o número de elementos

é menor do que 1, ou seja, terminamos depois de i passagens quando n

$$2 < 1$$

i

ou

$$n < 2i$$

ou

$$\log_2(n) < i.$$

Deste modo, para uma lista com n elementos, a procura binária não exige mais do que $\log_2(n)$ passagens. Em cada passagem efectuamos três comparações (uma comparação na instrução `if` e duas comparações para calcular a expressão que controla o ciclo `while`). Consequentemente, o número de comparações exigida pela procura binária é inferior ou igual a $3 \cdot \log_2(n)$.

No algoritmo de ordenação por borbulhamento, trocamos pares adjacentes de valores. O processo de ordenação consiste num certo número de passagens por todos os elementos da lista. O algoritmo da página 148 utiliza duas estruturas de repetição encadeadas. A estrutura exterior é realizada através de uma instrução `while`, e a interior com uma instrução `for`. Sendo n o número de elementos da lista, cada vez que o ciclo interior é executado, efectuam-se $n - 1$ comparações.

O ciclo exterior é executado até que todos os elementos estejam ordenados.

Na situação mais favorável, é executado apenas uma vez; na situação mais

156

CAPÍTULO 5. LISTAS

desfavorável é executado n vezes, neste caso, o número de comparações será $n \cdot (n - 1)$. O caso médio cai entre estes dois extremos, pelo que o número médio de comparações necessárias para a ordenação por borbulhamento será $n \cdot (n - 1)$

=

2

$$2 \cdot (n^2 - n).$$

No algoritmo de ordenação por selecção (apresentado na página 150), a operação básica é a selecção do menor elemento de uma lista. O algoritmo de ordenação por selecção é realizado através da utilização de dois ciclos for encadeados.

Para uma lista com n elementos, o ciclo exterior é executado $n - 1$ vezes; o número de vezes que o ciclo interior é executado depende do valor da variável i do ciclo exterior. A primeira vez será executado $n - 1$ vezes, a segunda vez $n - 2$ vezes e a última vez será executado apenas uma vez. Assim, o número de comparações exigidas pela ordenação por selecção é $(n - 1) + (n - 2) + \dots + 2 + 1$

Utilizando a fórmula para calcular a soma de uma progressão aritmética, obtemos o seguinte resultado para o número de comparações na ordenação por selecção:

(

1

(

$$n - 1) + 1$$

$$n - 1) \cdot$$

=

2

$$2 \cdot (n^2 - n).$$

5.7.1

A notação

ao do Omaiúsculo

A nossa preocupação com a eficiência está relacionada com problemas que envolvem um número muito grande de elementos. Se estivermos a procurar numa lista com cinco elementos, mesmo o

algoritmo menos eficiente nos resolve rapidamente o problema. Contudo, à medida que o número de elementos considerado cresce, o esforço necessário começa a diferir consideravelmente de algoritmo para algoritmo. Por exemplo, se a lista em que efectuamos a comparação tiver 100 000 000 de elementos, a procura sequencial requer uma média de 50 000 000

de comparações, ao passo que a procura binária requer, no máximo, 61 comparações!

Podemos exprimir uma aproximação da relação entre a quantidade de trabalho necessário e o número de elementos considerados, utilizando uma notação matemática conhecida por ordem de magnitude ou notação do Omaiúsculo (lido,

5.7. CONSIDERAC

, ~

OES SOBRE EFICIÊNCIA

157

ó maiúsculo9).

A notação do Omaiúsculo é usada em matemática para descrever o comportamento de uma função, normalmente em termos de outra função mais simples, quando o seu argumento tende para o infinito. A notação do Omaiúsculo pertence a uma classe de notações conhecidas por notações de Bachmann-Landau¹⁰

ou notações assintóticas. A notação do Omaiúsculo caracteriza funções de acordo com a sua taxa de crescimento, funções diferentes com a mesma taxa de crescimento podem ser representadas pela mesma notação do Omaiúsculo. Existem outras notações associadas à notação do Omaiúsculo, usando os símbolos o , Ω , ω e Θ para descrever outros tipos de taxas de crescimento.

Sejam $f(x)$ e $g(x)$ duas funções com o mesmo domínio definido sobre o conjunto dos números reais. Escrevemos $f(x) = O(g(x))$ se e só se existir uma constante positiva k , tal que para valores suficientemente grandes de x , $|f(x)| \leq k \cdot |g(x)|$, ou seja, se e só se existe um número real positivo k e um real x_0 tal que $|f(x)| \leq$

$k \cdot |g(x)|$ para todo o $x > x_0$.

A ordem de magnitude de uma função é igual à ordem do seu termo que cresce mais rapidamente em relação ao argumento da função. Por exemplo, a ordem de magnitude de $f(n) = n + n^2$ é n^2 uma vez que, para grandes valores de n , o valor de n^2 domina o valor de n (é tão importante face ao valor de n , que no cálculo do valor da função podemos praticamente desprezar este termo). Utilizando a notação do Omaiúsculo, a ordem de magnitude de $n^2 + n$ é $O(n^2)$.

Tendo em atenção esta discussão, podemos dizer que a procura binária é de ordem $O(\log_2(n))$, a procura sequencial é de ordem $O(n)$, e tanto a ordenação por borbulhamento como a ordenação por selecção são de ordem $O(n^2)$.

Os recursos consumidos por uma função não dependem apenas do algoritmo utilizado mas também do grau de dificuldade ou dimensão do problema a ser resolvido. Por exemplo, vimos que a procura binária exige menos recursos do que a procura sequencial. No entanto, estes recursos dependem da dimensão do problema em causa (do número de elementos da lista): certamente que a procura de um elemento numa lista de 5 elementos recorrendo à procura sequencial consome menos recursos do que a procura numa lista de 100 000 000 elementos utilizando a procura binária. O grau de dificuldade de um problema é tipicamente dado por um número que pode estar relacionado com o valor de um dos 9Do inglês, “Big-O”.

10Em honra aos matemáticos Edmund Landau (1877–1938) e Paul Bachmann (1837–1920).

158

CAPÍTULO 5. LISTAS

n

$\log_2(n)$

n^3

$6n$

$n!$

19

0.044

72.101

2.031×10^5

4.055×10^7

segundos

segundos

anos

anos

(Homo sapiens)

(tempo dos

dinossauros)

20

0.045

84.096

1.219×10^6

8.110×10^8

segundos

segundos

anos

anos

(início da vida

na terra)

21

0.046

97.351

7.312×10^6

1.703×10^{10}

segundos

segundos

anos

anos

(superior à idade

do universo)

22

0.047

1.866

4.387×10^7

segundos

minutos

anos

(tempo dos

dinossauros)

23

0.048

2.132

2.632×10^8

segundos

minutos

anos

24

0.048

2.422

1.579×10^9

segundos

minutos

anos

25

0.049

2.738

9.477×10^9

segundos

minutos

anos

(superior à idade

da terra)

26

0.049

3.079

5.686×10^{10}

segundos

minutos

anos

(superior à idade

do universo)

100

0.069

2.920

segundos

horas

500

0.094

15.208

segundos

dias

1000

0.105

121.667

segundos

dias

Tabela 5.2: Duração comparativa de algoritmos.

argumentos do problema (nos casos da procura e da ordenação, o número de elementos da lista) ou o grau de precisão exigido (como no caso do erro admitido na função para o cálculo da raiz quadrada apresentada na Secção 3.4.5), *etc.*

De modo obter uma noção das variações profundas relacionadas com as ordens de algumas funções, suponhamos que dispomos de um computador capaz de realizar 3 mil milhões de operações por segundo (um computador com um processador de 3GHz). Na Tabela 5.2 apresentamos a duração aproximada de alguns algoritmos com ordens de crescimento diferentes, para alguns valores do grau de dificuldade do problema. Notemos que para um problema com grau de dificuldade 19, se o seu crescimento for de ordem $O(6n)$, para que a

5.8. NOTAS FINAIS

159

termine nos nossos dias, esta teria que ter sido iniciada no período em que surgiu o Homo sapiens e para uma função com crescimento de ordem $O(n!)$ esta teria que ter sido iniciada no tempo dos dinossauros para estar concluída na actualidade. Por outro lado, ainda com base na Tabela 5.2, para um problema com grau de dificuldade 21, a resolução de um problema com ordem $O(\log_2(n))$ demoraria na ordem das 4.6 centésimas de segundo ao passo que a resolução de um problema com ordem $O(n!)$ requeria mais tempo do que a idade do universo.

5.8

Notas finais

Neste capítulo apresentámos o tipo lista, o qual é muito comum em programação, sendo tipicamente conhecido por tipo vector o tipo tabela. A lista é um tipo mutável, o que significa que podemos alterar destrutivamente os seus elementos.

Apresentámos alguns algoritmos de procura e de ordenação aplicados a listas.

Outros métodos de procura podem ser consultados em [Knuth, 1973b] e [Cormen et al., 2009].

Apresentámos uma primeira abordagem à análise da eficiência de um algoritmo através do estudo da ordem de crescimento e introdução da notação do Omaiúsculo. Para um estudo mais aprofundado da análise da eficiência de algoritmos, recomendamos a leitura de [Cormen et al., 2009], [Edmonds, 2008] ou

[McConnell, 2008].

5.9

Exercícios

5.1 Suponha que a operação `in` não existia em Python. Escreva uma função em Python, com o nome `pertence`, que recebe como argumentos uma lista e um inteiro e devolve `True`, se o inteiro está

armazenado na lista, e False, em caso contrário. Não pode usar um ciclo for pois este recorre à operação in. Por exemplo,

```
> > > pertence(3, [2, 3, 4])
```

True

```
> > > pertence(1, [2, 3, 4])
```

False

160

CAPÍTULO 5. LISTAS

5.2 Escreva uma função chamada substitui que recebe uma lista, lst, dois valores, velho e novo, e devolve a lista que resulta de substituir em lst todas as ocorrências de velho por novo. Por exemplo,

```
> > > substitui([1, 2, 3, 2, 4], 2, 'a')
```

```
[1, 'a', 3, 'a', 4]
```

5.3 Escreva uma função chamada posicoes_lista que recebe uma lista e um elemento, e devolve uma lista contendo todas as posições em que o elemento ocorre na lista. Por exemplo,

```
> > > posicoes_lista(['a', 2, 'b', 'a'], 'a')
```

```
[0, 3]
```

5.4 Escreva uma função chamada parte que recebe como argumentos uma lista, lst, e um elemento, e, e que devolve uma lista de dois elementos, contendo na primeira posição a lista com os elementos de lst menores que e, e na segunda posição a lista com os elementos de lst maiores ou iguais a e. Por exemplo,

```
> > > parte([2, 0, 12, 19, 5], 6)
```

```
[[2, 0, 5], [12, 19]]
```

```
> > > parte([7, 3, 4, 12], 3)
```

```
[[], [7, 3, 4, 12]]
```

5.5 Escreva um programa em Python que leia uma lista contendo inteiros e mude a ordem dos seus elementos de modo que estes

apareçam por ordem inversa.

5.6 Uma matriz é uma tabela bidimensional em que os seus elementos são referenciados pela linha e pela coluna em que se encontram. Uma matriz pode ser representada como uma lista cujos elementos são listas. Com base nesta representação, escreva uma função, chamada elemento matriz que recebe como argumentos uma matriz, uma linha e uma coluna e que devolve o elemento da matriz que se encontra na linha e coluna indicadas.

5.7 Considere uma matriz como definida no exercício anterior. Escreva uma

5.9. EXERCÍCIOS

161

2

3

1

4

5

6

7

8

1

2

5

4

3

6

7

8

1

2

4

3

5

6

7

8

1

2

3

4

5

6

7

8

1

2

3

4

5

6

7

8

Figura 5.12: Passos seguidos no quick sort.

função em Python que recebe uma matriz e que a escreve sob a forma

a12

...

... a1n

a21

a22

...

... a2n

...

...

...

an1

...

...

... ann

5.8 Considere, de novo, o conceito de matriz. Escreva uma função em Python que recebe como argumentos duas matrizes e devolve uma matriz correspondente ao produto das matrizes que são seus argumentos. Os elementos da matriz produto são dados por

n



$p_{ij} =$

$a_{ik}b_{kj}$

$k=1$

5.9 Um método de ordenação muito eficiente, chamado quick sort, consiste em considerar um dos elementos a ordenar (em geral, o primeiro elemento da lista), e dividir os restantes em dois grupos, um deles com os elementos menores e o outro com os elementos maiores que o elemento considerado.

Este é colocado entre os dois grupos, que são por sua vez ordenados utilizando quick sort. Por exemplo, os passos apresentados na Figura 5.12

correspondem à ordenação da lista

6
2
3
1
8
4
7
5
162

CAPÍTULO 5. LISTAS

utilizando quick sort (o elemento escolhido em cada lista representa-se a carregado).

Escreva uma função em Python para efectuar a ordenação de uma lista utilizando quick sort.

5.10 As funções de ordenação apresentadas neste capítulo correspondem a uma ordenação destrutiva, pois a lista original é destruída e substituída pela lista ordenada. Um processo alternativo consiste em criar uma lista com

índices, que representam as posições ordenadas dos elementos da lista.

Escreva uma função em Python para efectuar a ordenação por

selecção, criando uma lista com índices.

Capítulo 7

Recursão

ao e iteração

ao

‘Well, I’ll eat it,’ said Alice, ‘and if it makes me grow larger, I can reach the key; and if it makes me grow smaller, I can creep under the door: so either way I’ll get into the garden, and I don’t care which happens!’

She ate a little bit, and said anxiously to herself, ‘Which way? Which way?’

Lewis Carroll, *Alice’s Adventures in Wonderland* Nos capítulos anteriores considerámos alguns aspectos da programação, estudámos o modo de criar novas funções e utilizámos tipos estruturados de informação existentes em Python. Embora sejamos já capazes de escrever programas, o conhecimento que adquirimos ainda não é suficiente para podermos programar de um modo eficiente. Falta-nos saber que funções vale a pena definir e quais as consequências da execução de uma função.

Recordemos que a entidade básica subjacente à computação é o processo computacional. Na actividade de programação planeamos a sequência de acções a serem executadas por um programa. Para podermos desenvolver programas adequados a um dado fim é essencial que tenhamos uma compreensão clara dos processos computacionais gerados pelos diferentes tipos de funções. Este aspecto é abordado neste capítulo.

Uma função pode ser considerada como a especificação da evolução local de um processo computacional. Por evolução local entenda-se que a função define, em cada instante, o comportamento do processo computacional, ou seja, especifica

AO

como construir cada estágio do processo a partir do estágio anterior. Ao abordar processos computacionais, queremos estudar a evolução global do processo cuja evolução local é definida por uma função 1. Neste capítulo, apresentamos alguns padrões típicos da evolução de processos computacionais, estudando a ordem de grandeza do número de operações associadas e o “espaço” exigido pela evolução global do processo.

7.1

Recursão

ao linear

Começamos por considerar funções que geram processos que apresentam um padrão de evolução a que se chama recursão linear.

Consideremos a função factorial apresentada na Secção 6.1: def factorial (n):

if n == 0:

return 1

else:

return n * factorial(n - 1)

Vimos que durante a avaliação de uma expressão cujo operador é factorial, existem duas fases distintas: numa primeira fase, a execução da operação de multiplicação vai sendo sucessivamente adiada até se atingir o valor que corresponde à parte básica e numa segunda fase, as operações que foram adiadas são executadas.

Consideremos a definição da função potencia apresentada na Secção 6.3: def potencia (x, n):

if n == 0:

return 1

else:

```
return x * potencia(x, n - 1)
```

Durante a avaliação de uma expressão cujo operador é potencia, gera-se também um encadeamento de operações suspensas à espera do valor de outras expressões. Note-se que, na realidade, a função também define a evolução global do processo.

7.1. RECURSÃO

AO LINEAR

197

cujo operador é a própria potencia. Este encadeamento de operações pode ser representado do seguinte modo:

potencia(2, 8)

2 * potencia(2, 7)

2 (2 potencia(2, 6))

2 (2 (2 * potencia(2,5)))

2 (2 (2 (2 potencia(2,4))))

2 (2 (2 (2 (2 * potencia(2, 3)))))

2 (2 (2 (2 (2 (2 potencia(2,2))))) 2 (2 (2 (2 (2 (2 * potencia(2,1))))) 2 (2 (2 (2 (2 (2 (2 potencia(2, 0))))) 2 (2 (2 (2 (2 (2 1))))) 2 (2 (2 (2 (2 (2 * 2)))))

2 (2 (2 (2 (2 4))))

2 (2 (2 (2 * 8)))

2 (2 (2 16)))

2 (2 (2 * 32))

2 (2 64)

2 * 128

256

No processo para o cálculo de uma potência existem também duas fases distintas: na primeira, a execução da operação de

multiplicação vai sendo sucessivamente adiada até se atingir o valor que corresponde à parte básica da função potencia, na segunda, as operações que foram adiadas são executadas.

Como terceiro exemplo, consideremos uma função, soma elementos, para calcular a soma dos elementos de uma lista2. Podemos escrever a seguinte função em Python para calcular a soma dos elementos de uma lista: def soma_elementos (l):

```
if l == []:
```

```
    return 0
```

```
else:
```

```
    return l[0] + soma_elementos(l[1:])
```

2Consideramos que a lista contém números, pelo que não faremos qualquer teste ao seu conteúdo.

198

CAPÍTULO 7. RECURSÃO

AO E ITERAC

~
,

AO

Ou seja, se a lista não tem elementos, a soma dos seus elementos é zero, em caso contrário, a soma dos seus elementos corresponde ao resultado de somar o primeiro elemento da lista ao resultado de calcular a soma dos restantes elementos da lista.

Podemos agora considerar a forma do processo gerado pela função soma elementos:

```
soma_elementos([1, 2, 3, 4])
```

```
1 + soma_elementos([2, 3, 4])
```

```
1 + (2 + soma_elementos([3, 4]))
```

```
1 + (2 + (3 + soma_elementos([4])))
```

```
1 + (2 + (3 + (4 + soma_elementos([]))))
```


$$1 + (2 + (3 + (4 + 0)))$$

$$1 + (2 + (3 + 4))$$

$$1 + (2 + 7)$$

$$1 + 9$$

$$10$$

Voltamos a deparar-nos com um processo que apresenta duas fases distintas; numa primeira fase verifica-se uma expansão, devido à existência de operações cuja aplicação vai sendo sucessivamente adiada, seguida por uma fase de contracção na qual as operações que foram adiadas são executadas.

Embora as três funções que apresentámos tenham finalidades distintas, todas elas geram processos computacionais que têm um comportamento semelhante: são caracterizados por uma fase de expansão, devido à existência de operações adiadas, seguida por uma fase de contracção em que essas operações são executadas. Este padrão de evolução de um processo é muito comum em programação e tem o nome de processo recursivo.

Num processo recursivo existe uma fase de expansão correspondente à construção de uma cadeia de operações adiadas, seguida por uma fase de contracção correspondente à execução dessas operações. A informação acerca destas operações adiadas é mantida internamente pelo Python.

Nos três casos que apresentámos, o número de operações adiadas cresce linearmente com um determinado valor. Este valor pode corresponder a um dos

7.2. ITERAC

~
,

AO LINEAR

199

parâmetros da função (como é o caso do inteiro para o qual se está a calcular o factorial, do expoente relativamente à potência e do número de elementos da lista), mas pode também corresponder a outras coisas.

A um processo recursivo que cresce linearmente com um valor dá-se o nome de processo recursivo linear.

7.2

Iteraç~

ao linear

Nesta secç~ao consideramos de novo os problemas discutidos na secç~ao anterior e apresentamos a sua soluç~ao através de func~oes que geram processos que apresentam um padr~ao de evoluç~ao a que se chama iteraç~ao linear. Após a análise dos processos gerados por estas func~oes, caracterizamos a iteraç~ao linear.

Suponhamos que somos postos perante o problema de calcular manualmente o factorial de um número, por exemplo 8!. O nosso processo de cálculo seria de novo bem diferente do processo recursivo linear apresentado na Secç~ao 7.1.

Começaríamos a efectuar multiplicaç~oes, por exemplo da esquerda para a direita, em lugar de adiar a sua realizaç~ao: o nosso processo seguiria o raciocínio: oito vezes sete 56, vezes seis 336, vezes cinco 1 680, vezes quatro 6 720, vezes três 20 160, vezes dois 40 320. Durante o nosso processo temos de nos lembrar do valor do produto acumulado e, adicionalmente, de qual o próximo número a utilizar na multiplicaç~ao.

O nosso processo de cálculo pode ser traduzido pela seguinte func~ao em Python, na qual `prod_ac` representa o produto acumulado dos números que já multiplicámos e `prox` representa o próximo número a ser multiplicado: `def factorial_aux (prod_ac, prox):`

```
if prox == 0:
```

```
    return prod_ac
```

```
else:
```

```
    return factorial_aux (prod_ac * prox, prox - 1)
```

Com base nesta func~ao, podemos agora escrever a seguinte func~ao para o cálculo de factorial, a qual esconde do exterior a utilizaç~ao das variáveis `prod_ac` e `prox`:

```
def factorial (n):
```

CAPÍTULO 7. RECURSÃO

AO E ITERACÃO

AO

```
def factorial_aux (prod_ac, prox):
```

```
    if prox == 0:
```

```
        return prod_ac
```

```
    else:
```

```
        return factorial_aux (prod_ac * prox, prox - 1)
```

Durante a avaliação de uma expressão cujo operador é factorial, não se gera um encadeamento de operações suspensas e, consequentemente, o processo não se expande nem se contrai:

```
factorial(8)
```

```
factorial_aux(1, 8)
```

```
factorial_aux(8, 7)
```

```
factorial_aux(56, 6)
```

```
factorial_aux(336, 5)
```

```
factorial_aux(1680, 4)
```

```
factorial_aux(6720, 3)
```

```
factorial_aux(20160, 2)
```

```
factorial_aux(40320, 1)
```

```
factorial_aux(40320, 0)
```

```
40320
```

Adicionalmente, em cada instante, possuímos toda a informação necessária para saber o que já fizemos (o produto acumulado, prod ac) e o que ainda nos falta fazer (o número de multiplicações a realizar,

prox), informação essa que não existe explicitamente no processo recursivo linear. Isto significa que podemos interromper o processo computacional em qualquer instante e recomeçá-lo mais tarde, utilizando a informação disponível no instante em que este foi interrompido.

O raciocínio que utilizámos nesta segunda versão da função factorial é directamente aplicado no desenvolvimento de uma versão desta função recorrendo à programação imperativa:

```
def factorial (n):
```

7.2. ITERAC

~
,

AO LINEAR

201

```
prod_ac = 1
```

```
for prox in range(1, n + 1):
```

```
    prod_ac = prod_ac * prox
```

```
return prod_ac
```

Suponhamos agora que somos postos perante o problema de calcular manualmente uma potência de um número, por exemplo 28. O nosso processo de cálculo seria bem diferente do processo recursivo linear. Começaríamos a efectuar multiplicações, em lugar de adiar a sua realização: o nosso processo seguiria o seguinte raciocínio: dois vezes dois 4, vezes dois 8, vezes dois 16, e assim sucessivamente. Durante a evolução do nosso processo, temos de nos lembrar do valor parcial que já calculámos para a potência e, adicionalmente, de quantas vezes já efectuámos a multiplicação.

Em resumo, este processo de cálculo corresponde a manter um valor para o produto acumulado num dado instante e, simultaneamente, saber quantas vezes ainda nos falta multiplicar. Cada vez que efectuamos uma multiplicação, o valor do produto acumulado altera-se (pois ele é multiplicado pela base) e o número de vezes que ainda temos de multiplicar diminui em uma unidade. O processo termina quando não falta multiplicar nenhuma vez.

A seguinte função simula o método de cálculo que acabámos de descrever. Na função potencia_aux, o nome n_mult corresponde ao número de multiplicações que temos de executar, e o nome prod_ac corresponde ao produto acumulado.

Em cada passo, actualizamos simultaneamente os valores do número de multiplicações a efectuar e do produto acumulado. Ao iniciar o cálculo, estabelecemos que o produto acumulado é a base (x), e o número de multiplicações que nos falta efectuar corresponde ao expoente (n) menos um (embora não tenhamos feito qualquer operação, já contamos com a base no produto acumulado):

```
def potencia_aux (x, n_mult, prod_ac):
```

```
    if n_mult == 0:
```

```
        return prod_ac
```

```
    else:
```

3Em alternativa, podíamos estabelecer como valores iniciais do produto acumulado e do número de multiplicações

oes a efectuar, 1 e o expoente, respectivamente.

202

CAPÍTULO 7. RECURSÃO

AO E ITERACÃO

~

AO

```
return potencia_aux (x, n_mult - 1, x * prod_ac) return potencia_aux (x, n - 1, x)
```

Podemos agora representar a sequência de expressões que são avaliadas pelo Python, envolvendo estas duas funções, na sequência da avaliação de potencia(2, 8):

```
potencia(2, 8)
```

```
potencia_aux(2, 7, 2)
```

potencia_aux(2, 6, 4)

potencia_aux(2, 5, 8)

potencia_aux(2, 4, 16)

potencia_aux(2, 3, 32)

potencia_aux(2, 2, 64)

potencia_aux(2, 1, 128)

potencia_aux(2, 0, 256)

256

Este processo apresenta um padrão muito diferente do encontrado no processo recursivo linear. Aqui não existem operações suspensas, e o processo não se expande nem se contrai.

Adicionalmente, em cada instante, possuímos toda a informação necessária para saber o que já fizemos (o produto acumulado) e o que ainda nos falta fazer (o número de multiplicações a realizar), informação essa que não existe explicitamente no processo recursivo linear. Isto significa que podemos interromper o processo computacional em qualquer instante e recomeçá-lo mais tarde, utilizando a informação disponível no instante em que este foi interrompido.

Novamente, podemos aplicar directamente o nosso raciocínio ao desenvolvimento de uma versão da função potencia recorrendo à programação imperativa: def potencia (x, n):

```
prod_ac = 1
```

```
for n_mult in range(n, 0, -1):
```

```
    prod_ac = prod_ac * x
```

```
return prod_ac
```

7.2. ITERAC

~
s

AO LINEAR

Finalmente, para calcular a soma dos elementos de uma lista, podemos utilizar um raciocínio semelhante aos anteriores. Teremos que manter um registo das somas dos elementos que já considerámos, o qual é representado pela variável `soma`, e quando a lista for vazia, o valor de `soma` corresponde à soma dos elementos da lista.

Este processo é traduzido pela seguinte função: `def soma_elementos(l):`

```
def soma_elementos_aux(l, soma):

    if l == []:

        return soma

    else:

        return soma_elementos_aux(l[1:], l[0] + soma)

return soma_elementos_aux(l, 0)
```

Tal como nos dois casos anteriores, a avaliação da função `soma_elementos` origina um processo que não se expande nem se contrai. Em cada instante, toda a informação que é manipulada pelo processo está explicitamente disponível: a lista que está a ser considerada nesse instante e a soma dos elementos já encontrados:

```
soma-elementos([1, 2, 3, 4])
soma-elementos-aux([1, 2, 3, 4], 0)
soma-elementos-aux([2, 3, 4], 1)
soma-elementos-aux([3, 4], 3)
soma-elementos-aux([4], 6)
soma-elementos-aux([], 10)
```

10

Usando a programação imperativa, no nosso raciocínio origina a seguinte versão da função `soma_elementos`:

```
def soma_elementos(l):
```

```
soma = 0
```

```
for e in l:
```

```
204
```

CAPÍTULO 7. RECURSÃO

AO E ITERAÇÃO

```
AO
```

```
soma = soma + e
```

```
return soma
```

Novamente, embora as três funções que apresentámos tenham finalidades distintas, todas elas geram processos computacionais que têm um comportamento semelhante: os processos não se expandem nem se contraem. Eles são caracterizados por um certo número de variáveis que fornecem uma descrição completa do estado da computação em cada instante. Esta situação não acontece num processo recursivo no qual o número de operações suspensas é “escondido” pelo Python. O que fizemos nas três funções anteriores foi tornar explícita a informação que estava escondida no processo recursivo correspondente. O padrão de evolução de um processo que descrevemos nesta secção é também muito comum em programação, e tem o nome de processo iterativo. Recorrendo ao Wikcionário⁴, a palavra “iterativo” tem o seguinte significado “Diz-se do processo que se repete diversas vezes para se chegar a um resultado e a cada vez gera um resultado parcial que será usado na vez seguinte”.

Um processo iterativo é caracterizado por um certo número de variáveis, chamadas variáveis de estado, juntamente com uma regra que especifica como actualizar-las. Estas variáveis fornecem uma descrição completa do estado da computação em cada momento. Um processo iterativo não se expande nem se contrai.

Nos nossos exemplos, o número de operações efectuadas sobre as variáveis de estado cresce linearmente com uma grandeza associada à função (o inteiro para o qual se está a calcular o factorial, o expoente no caso da potência e o o número de elementos da lista).

A um processo iterativo cujo número de operações cresce linearmente

com um valor dá-se o nome de processo iterativo linear.

7.3

Recurs~

ao em processos e em funç~

oes

Da discuss~ao apresentada no Capítulo 6 e na Secç~ao 7.1 podemos concluir que a palavra “recurs~ao” tem dois significados distintos, conforme se refere à recurs~ao em funç~oes ou à recurs~ao em processos. A recurs~ao em funç~oes refere-se à de-4http://pt.wiktionary.org/wiki.

7.4. RECURS ~

AO EM ÁRVORE

205

finiç~ao da funç~ao em termos de si própria, ao passo que a recurs~ao em processos refere-se ao padr~ao de evoluç~ao do processo.

Podemos também concluir que a evoluç~ao de processos pode ser classificada como uma evoluç~ao recursiva ou como uma evoluç~ao iterativa: 1. Um processo recursivo é caracterizado por uma fase de expans~ao (correspondente à construc~ao de uma cadeia de operaç~oes adiadas) seguida de uma fase de contracç~ao (correspondente à execuç~ao dessas operaç~oes). O

computador mantém informaç~ao “escondida” que regista o ponto onde está o processo na cadeia de operaç~oes adiadas.

2. Um processo iterativo ão cresce nem se contrai. Este é caracterizado por um conjunto de variáveis de estado e um conjunto de regras que definem como estas variáveis evoluem. As variáveis de estado fornecem uma descriç~ao completa do estado do processo em cada ponto.

Repare-se que uma funç~ao recursiva tanto pode gerar um processo recursivo como um processo iterativo. Por exemplo, tanto funç~ao potencia da Secç~ao 7.1

como a funç~ao potencia aux da Secç~ao 7.2 s~ao funç~oes recursivas (s~ao definidas em termos de si próprias), no entanto a primeira gera

um processo recursivo, e a segunda gera um processo iterativo.

7.4

Recurs~

ao em árvore

Nesta secção vamos considerar um outro padrão da evolução de processos que também é muito comum em programação, a recursão em árvore.

7.4.1

Os números de Fibonacci

Para ilustrar a recursão em árvore vamos considerar uma sequência de números descoberta no século XIII pelo matemático italiano Leonardo Fibonacci (c.

1170–c. 1250), também conhecido por Leonardo de Pisa, ao tentar resolver o seguinte problema:

“Quantos casais de coelhos podem ser produzidos a partir de um único casal durante um ano se cada casal originar um novo casal em cada mês, o qual se torna fértil a partir do segundo mês; e n~

ao ocorrerem mortes.”

206

CAPÍTULO 7. RECURS~

AO E ITERAC

~
s

AO

Fibonacci chegou à conclusão que a evolução do número de casais de coelhos era ditada pela seguinte sequência: 0, 1, 1, 2, 3, 5, 8, 13, 21, . . . Nesta sequência, conhecida por sequência de Fibonacci, cada termo, excepto os dois primeiros, é a soma dos dois anteriores. Os dois primeiros termos são respectivamente 0 e 1. Os números da sequência de Fibonacci são conhecidos por números de Fibonacci, e podem ser descritos através da seguinte definição:

$\{ 0$

se $n = 0$

$\text{fib}(n) =$

1

se $n = 1$

$\text{fib}(n - 1) + \text{fib}(n - 2)$ se $n > 1$

Suponhamos que desejávamos escrever uma função em Python para calcular os números de Fibonacci. Com base na definição anterior podemos produzir a seguinte função:

```
def fib (n):
```

```
    if n == 0:
```

```
        return 0
```

```
    elif n == 1:
```

```
        return 1
```

```
    else:
```

```
        return fib(n - 2) + fib(n - 1)
```

Vejamos agora qual a forma do processo que é gerado para o cálculo de um número de Fibonacci, por exemplo, $\text{fib}(5)$. Assumindo que as sub-expressões numa expressão composta são avaliadas da esquerda para a direita, obtemos a seguinte evolução:

$\text{fib}(5)$

$\text{fib}(4) + \text{fib}(3)$

$(\text{fib}(3) + \text{fib}(2)) + \text{fib}(3)$

$((\text{fib}(2) + \text{fib}(1)) + \text{fib}(2)) + \text{fib}(3)$

$((\text{fib}(1) + \text{fib}(0)) + \text{fib}(1)) + \text{fib}(2)) + \text{fib}(3) (((1 + 0) + \text{fib}(1)) + \text{fib}(2)) + \text{fib}(3)$

$((1 + 1) + \text{fib}(2)) + \text{fib}(3)$

$(2 + (\text{fib}(1) + \text{fib}(0)) + \text{fib}(3)$

$(2 + (1 + 0)) + \text{fib}(3)$

No caso de uma ordem diferente de avaliação, a “forma” do processo é semelhante.

7.4. RECURS ~

AO EM ÁRVORE

207

$3 + \text{fib}(3)$

$3 + (\text{fib}(2) + \text{fib}(1))$

$3 + ((\text{fib}(1) + \text{fib}(0)) + \text{fib}(1))$

$3 + ((1 + 0) + \text{fib}(1))$

$3 + (1 + \text{fib}(1))$

$3 + (1 + 1)$

$3 + 2$

5

Ao analisarmos a forma do processo anterior, verificamos que esta não corresponde a nenhum dos padrões já estudados. No entanto, este apresenta um comportamento que se assemelha ao processo recursivo. Tem fases de crescimento, originadas por operações adiadas, seguidas por fases de contracção em que algumas das operações adiadas são executadas.

Ao contrário do que acontece com o processo recursivo linear, estamos perante a existência de múltiplas fases de crescimento e de contracção que são originadas pela dupla recursão que existe na função `fib` (esta refere-se duas vezes a si própria). A este tipo de evolução de um processo dá-se o nome de recursão em árvore. Esta designação deve-se ao facto de a evolução do processo ter a forma de

uma árvore. Cada avaliação da expressão composta cujo operador é a função fib dá origem a duas avaliações (dois ramos de uma árvore), excepto para os dois últimos valores. Na Figura 7.1 apresentamos o resultado da árvore gerada durante o cálculo de fib(5).

Estudando a evolução do processo anterior, verificamos facilmente que este é muito ineficiente, pois existem muitos cálculos que são repetidos múltiplas vezes. Para além disso, o processo utiliza um número de passos que não cresce linearmente com o valor de n. Demonstra-se que este número cresce exponencialmente com o valor de n.

Para evitar os cálculos repetidos, vamos agora calcular os números de Fibonacci através de um processo iterativo. Sabemos que cada termo da sequência de Fibonacci, excepto os dois primeiros, é a soma dos dois anteriores. Para calcular um dos termos da sequência teremos, pois, de saber os dois termos anteriores.

Suponhamos que, no cálculo de um termo genérico f_n , os dois termos anteriores são designados por f_{n-2} e f_{n-1} . Neste caso, o próximo número de Fibonacci será dado por $f_n = f_{n-2} + f_{n-1}$ e a partir de agora os dois últimos termos são f_{n-1}

e f_n . Podemos agora, usando o mesmo raciocínio, calcular o próximo termo,

208

CAPÍTULO 7. RECURSÃO

AO E ITERACÃO

AO

fib(5)

~
5

†

† ○

†

○



fib(4)

fib(3)



fib(3)

fib(2)

fib(2)

fib(1)



fib(2)

fib(1)

fib(1)

fib(0)

fib(1)

fib(0)



fib(1)

fib(0)

1

0

1

1

0

1

0

1

Figura 7.1: Árvore gerada durante o cálculo de fib(5).

$fib(n+1)$. Para completarmos o nosso processo teremos de explicitar o número de vezes que temos de efectuar estas operações.

Assim, podemos escrever a seguinte função que traduz o nosso processo de raciocínio. Esta função recebe os dois últimos termos da sequência ($fib(n-2)$ e $fib(n-1)$) e o número de operações que ainda temos de efectuar (cont).

```
def fib_aux (f_n_2, f_n_1, cont):
```

```
    if cont == 0:
```

```
        return f_n_2 + f_n_1
```

```
    else:
```

```
        return fib_aux(f_n_1, f_n_2 + f_n_1, cont - 1)
```

Para utilizar esta função devemos indicar os dois primeiros termos da sequência de Fibonacci:

```
def fib (n):
```

```
    if n == 0:
```

```
        return 0
```

7.4. RECURS ~

AO EM ÁRVORE


```
elif n == 1:
```

```
    return 1
```

```
else:
```

```
    return fib_aux(0, 1, n - 2)
```

Esta função origina um processo iterativo linear em que não há duplicações de cálculos:

```
fib(5)
```

```
fib_aux(0, 1, 3)
```

```
fib_aux(1, 1, 2)
```

```
fib_aux(1, 2, 1)
```

```
fib_aux(2, 3, 0)
```

```
5
```

Em programação imperativa, a função fib poderá ser definida do seguinte modo: def fib (n):

```
if n == 0:
```

```
    return 0
```

```
elif n == 1:
```

```
    return 1
```

```
else:
```

```
    f_n_2 = 0
```

```
    f_n_1 = 1
```

```
    cont = 0
```

```
    while cont < n - 1:
```

```
        f_n_2, f_n_1 = f_n_1, f_n_2 + f_n_1
```

```
cont = cont + 1
```

```
return f_n_1
```

Deste exemplo não devemos concluir que a recursão em árvore é um processo inútil. Na próxima seção apresentamos um outro problema para o qual a recursão em árvore corresponde ao método ideal para a sua resolução.

210

CAPÍTULO 7. RECURSÃO

AO E ITERAC

AO

~
,

*

Figura 7.2: Torre de Hanói com três discos.

7.4.2

A torre de Hanói

Apresentamos um programa em Python para a solução de um puzzle chamado a Torre de Hanói. A Torre de Hanói é constituída por 3 postes verticais, nos quais podem ser colocados discos de diâmetros diferentes, furados no centro, variando o número de discos de puzzle para puzzle. O puzzle inicia-se com todos os discos num dos postes (tipicamente, o poste da esquerda), com o disco menor no topo e com os discos ordenados, de cima para baixo, por ordem crescente dos respectivos diâmetros, e a finalidade é movimentar todos os discos para um outro poste (tipicamente, o poste da direita), também ordenados por ordem crescente dos respectivos diâmetros, de acordo com as seguintes regras: (1) apenas se pode movimentar um disco de cada vez; (2) em cada poste, apenas se pode movimentar o disco de cima; (3) nunca se pode colocar um disco sobre outro de diâmetro menor.

Este puzzle está associado a uma lenda, segundo a qual alguns monges num mosteiro perto de Hanói estão a tentar resolver um destes puzzles com 64 discos, e no dia em que o completarem será o fim do mundo. Não nos devemos preocupar com o fim do mundo, pois se os

monges apenas efectuarem movimentos perfeitos à taxa de 1 movimento por segundo, demorarão perto de mil milhões de anos para o resolver6.

Na Figura 7.2 apresentamos um exemplo das configurações inicial e final para a Torre de Hanói com três discos.

Suponhamos então que pretendíamos escrever um programa para resolver o puzzle da Torre de Hanói para um número n de discos (o valor de n será fornecido pelo utilizador). Para resolver o puzzle da Torre de Hanói com n discos ($n > 1$), teremos de efectuar basicamente três passos: 1. Movimentar $n - 1$ discos do poste da esquerda para o poste do centro 6Ver [Raphael, 1976], página 80.

7.4. RECURS ~

AO EM ÁRVORE

211

*

*

*

Figura 7.3: Solução da Torre de Hanói com três discos.

(utilizado como poste auxiliar);

2. Movimentar o disco do poste da esquerda para o poste da direita; 3. Mvimentar os $n - 1$ discos do poste do centro para o poste da direita.

Estes passos encontram-se representados na Figura 7.3 para o caso de $n = 3$.

Com este método conseguimos reduzir o problema de movimentar n discos ao problema de movimentar $n - 1$ discos, ou seja, o problema da movimentação de n discos foi descrito em termos do mesmo problema, mas tendo um disco a menos. Temos aqui um exemplo típico de uma solução recursiva. Quando n for igual a 1, o problema é resolvido trivialmente, movendo o disco da origem para o destino.

Como primeira aproximação, podemos escrever a função mova para movimentar n discos. Esta função tem como argumentos o número de discos a mover e uma indicação de quais os postes de origem e destino dos discos, bem como qual o poste que deve ser usado como

poste auxiliar: def mova(n, origem, destino, aux):

if n == 1:

mova_disco(origem, destino)

212

CAPÍTULO 7. RECURS ~

AO E ITERAC

, ~

AO

mova(3, 'E', 'D', 'C')

mova(2, 'E', 'C', 'D')

*

mova disco('E', 'D')

*

mova(2, 'C', 'D', 'E')

*

Figura 7.4: Subproblemas gerados por mova(3, 'E', 'D', 'C').

else:

mova(n-1, origem, aux, destino)

mova_disco(origem, destino)

mova(n-1, aux, destino, origem)

Antes de continuar, convém observar que a instrução composta que corresponde ao else é constituída por três instruções. Dentro destas instruções, as duas utilizações da função mova têm os argumentos correspondentes aos postes, por ordem diferente, o que corresponde a resolver dois outros puzzles em que os postes de origem, de destino e auxiliares são diferentes. Na Figura 7.4 apresentamos as três expressões que são originadas por mova(3, 'E', 'D', 'C'), ou seja, mova três discos do poste da esquerda para o poste da direita,

utilizando o poste do centro como poste auxiliar, bem como uma representação dos diferentes subproblemas que estas resolvem.

Esta função reflecte o desenvolvimento do topo para a base: o primeiro passo para a solução de um problema consiste na identificação dos subproblemas que o constituem, bem como a determinação da sua inter-relação. Escreve-se então uma primeira aproximação da solução em termos destes subproblemas. No nosso

7.4. RECURSÃO

AO EM ÁRVORE

213

exemplo, o problema da movimentação de n discos foi descrito em termos de dois subproblemas: o problema da movimentação de um disco e o problema da movimentação de $n - 1$ discos, e daí a solução recursiva.

Podemos agora escrever a seguinte função que fornece a solução para o puzzle da Torre de Hanói para um número arbitrário de discos:

```
def hanoi ():
```

```
def mova(n, origem, destino, aux):
```

```
def mova_disco (de, para):
```

```
print(de, '->', para)
```

```
if n == 1:
```

```
mova_disco(origem, destino)
```

```
else:
```

```
mova(n-1, origem, aux, destino)
```

```
mova_disco(origem, destino)
```

```
mova(n-1, aux, destino, origem)
```

```
n = eval(input('Quantos discos deseja considerar?\n? '))
```

```
print('Solução
```

```
ao do puzzle:')
```

mova(n, 'E', 'D', 'C')

Esta função permite originar a seguinte interacção:

> > > hanoi()

Quantos discos deseja considerar?

? 3

Solução

ao do puzzle:

E -> D

E -> C

D -> C

E -> D

C -> E

C -> D

E -> D

214

CAPÍTULO 7. RECURSÃO

AO E ITERAC

~
,

AO

torre-de-hanoi()

mova(3, 'E', 'D', 'C')

†

† ○

†



mov a(2, 'E', 'C', 'D')

mov a disco('E', 'D')

mov a(2, 'C', 'D', 'E')





move(1, 'E', 'D', 'C')

move(1, 'D', 'C', 'E')

move(1, 'C', 'E', 'D')

move(1, 'E', 'D', 'C')

move disco('E', 'C')

move disco('C', 'D')

move disco('E', 'D')

move disco('D', 'C')

move disco('C', 'E')

move disco('E', 'D')

E -> D

E -> C

D -> C

E -> D

C -> E

C -> D

E -> D

Figura 7.5: Árvore gerada por `torre-de-hanoi()` com três discos Na Figura 7.5 apresentamos a árvore gerada pela execução da função `hanoi()` quando o utilizador fornece o valor 3 para o número de discos. Nesta figura apresentam-se dentro de um rectângulo os valores que o Python escreve.

Estamos de novo perante uma função recursiva que origina um processo recursivo em árvore. Este exemplo mostra como a recursão pode dar origem a funções que são fáceis de escrever e de compreender.

7.5

Consideraç~

oes sobre eficiência

Os exemplos apresentados neste capítulo mostram que os processos gerados por funções podem diferir drasticamente quanto à taxa a que consomem recursos computacionais. Assim, um dos aspectos que vamos ter de levar em linha de

7.5. CONSIDERAC

~
,

OES SOBRE EFICI ÊNCIA

215

conta quando escrevemos programas é a minimização dos recursos computacionais consumidos. Os recursos computacionais que aqui vamos considerar são o tempo e o espaço. O tempo diz respeito ao tempo que o nosso programa demora a executar, e o espaço diz respeito ao espaço de memória do computador usado pelo nosso programa.

Desta discussão e dos exemplos apresentados neste capítulo, podemos concluir que os processos recursivos lineares têm ordem $O(n)$, quer para o espaço, quer para o tempo, e que os processos iterativos lineares têm ordem $O(1)$ para o espaço e $O(n)$ para o tempo.

Tendo em atenção as preocupações sobre os recursos consumidos por um processo, apresentamos uma alternativa para o cálculo de potência, a qual gera um processo com ordem de crescimento inferior ao que apresentámos. Para compreender o novo método de cálculo de uma potência, repare-se que podemos definir potência, do seguinte modo:

$$x^n = \begin{cases} x & \text{se } n = 1 \\ x \cdot (x^{n-1}) & \text{se } n \text{ for ímpar} \\ (x^{n/2})^2 & \text{se } n \text{ for par} \end{cases}$$

o que nos leva a escrever a seguinte função para o cálculo da potência: `def potencia_rapida(x, n):`

```
def potencia_rapida(x, n):
    if n == 1:
        return x
    elif impar(n):
        return x * potencia_rapida(x, n - 1)
    else:
        return quadrado(potencia_rapida(x, n // 2))
```

Nesta função, `quadrado` e `impar` correspondem às funções: `def quadrado(x):`

```
return x * x
```

```
def impar (x):
```

```
return (x % 2) == 1
```

216

CAPÍTULO 7. RECURS ~

AO E ITERAC

~
,

AO

Com este processo de cálculo, para calcular x^{2n} precisamos apenas de mais uma multiplicação do que as que são necessárias para calcular x^n . Geramos assim um processo cuja ordem temporal é $O(\log_2(n))$. Para apreciar a vantagem desta função, notemos que, para calcular uma potência de expoente 1 000, a função potencia necessita de 1 000 multiplicações, ao passo que a função potencia rápida apenas necessita de 14.

7.6

Notas finais

Neste capítulo apresentamos as motivações para estudar os processos gerados por procedimentos e caracterizamos alguns destes processos, nomeadamente os processos recursivos lineares, iterativos lineares e recursivos em árvore. Definimos a noção de ordem de crescimento e comparámos as taxas a que alguns processos consomem recursos.

7.7

Exercícios

7.1 Considere a afirmação “as funções que geram processos iterativos são sempre melhores que as funções que geram processos recursivos”. Tendo em atenção os exemplos apresentados neste capítulo, comente-a relativamente aos seguintes aspectos: (a) memória ocupada; (b) tempo gasto; (c) facilidade de escrita e leitura da função.

7.2 Considere a seguinte função:

```

def m_p (x, y):

def m_p_a(z):

if z == 0:

return 1

else:

return m_p_a(z - 1) * x

return m_p_a(y)

```

a) Apresente a evoluç~ao do processo na avaliaç~ao de $m_p(2, 4)$.

7.7. EXERCÍCIOS

217

b) A funç~ao gera um processo recursivo ou iterativo? Justifique a sua resposta.

c) Se a funç~ao gerar um processo recursivo, escreva uma funç~ao equivalente que gere um processo iterativo; se a funç~ao gerar um processo iterativo, escreva uma funç~ao equivalente que gere um processo recursivo.

7.3 Escreva uma funç~ao em Python para calcular o valor da funç~ao de Acker-mann7:

$$\begin{aligned}
 & \left\{ \begin{array}{l} n + 1 \\ \text{se } m = 0 \\ A(m, n) = \\ A(m - 1, 1) \\ \text{se } m > 0 \text{ e } n = 0 \\ \vdots \\ A(m - 1, A(m, n - 1)) \text{ se } m > 0 \text{ e } n > 0 \end{array} \right.
 \end{aligned}$$

Siga o processo gerado pelo cálculo de $A(2, 2)$.

7.4 Suponha que em Python não existia o operador de multiplicação, $*$, mas que existiam os operadores de adição, $+$, e divisão inteira, $//$. O produto de dois números inteiros positivos pode ser definido através da seguinte fórmula:

$$\begin{aligned} & \text{if } x = 0 \\ & \quad \text{return } 0 \\ & \text{else} \\ & \quad \text{return } x + \text{produto}(x - 1, y) \end{aligned}$$

a) Defina a função produto que calcula o produto de dois números inteiros positivos, de acordo com esta definição.

b) A sua função é recursiva? Justifique.

c) A sua função gera um processo recursivo ou iterativo? Justifique a sua resposta.

d) Se o processo gerado era iterativo, transforme a função, de forma a que esta gere um processo recursivo. Se o processo gerado era recursivo, transforme a função, de forma a que esta gere um processo iterativo.

7.5 O número de combinações de m objectos n a n pode ser dado pela seguinte 7Tal como apresentada em [Machtey and Young, 1978], página 24.

AO

fórmula:

```
def C(m, n):
    if n < 0:
        return 0
    if m < 0:
        return 0
    if m == n:
        return 1
    if m < n:
        return C(m, n-1) + C(m-1, n-1)
    if m > n:
        return C(m-1, n) + C(m-1, n-1)
```

a) Escreva uma função em Python para calcular o número de combinações de m objectos n a n.

b) Que tipo de processo é gerado pela sua função?

Capítulo 8

Ficheiros

‘They’re putting down their names,’ the Gryphon whispe-red in reply, ‘for fear they forget them before the end of the trial.’

Lewis Carroll, Alice's Adventures in Wonderland Os programas que temos desenvolvido até aqui têm uma única fonte de dados, o teclado, e um único destino para os seus resultados, o ecrã. Para além disso, quando o programa termina, os dados usados e produzidos pelo programa desaparecem.

A maior parte dos programas desenvolvidos na vida real têm múltiplas fontes de dados e múltiplos destinos para os seus resultados. Muitos dos dados dos programas utilizados na vida real são persistentes, no sentido em que eles existem, independentemente da execução do programa, armazenados no disco do computador, numa memória USB ou num CD. A estrutura tipicamente utilizada para armazenar esta informação é chamada um ficheiro.

Um ficheiro¹ é um tipo estruturado de informação constituído por uma sequência de elementos, todos do mesmo tipo. Nos ficheiros que consideramos neste capítulo, os elementos são acedidos sequencialmente, ou seja, para aceder ao n -ésimo elemento do ficheiro, teremos primeiro de aceder aos $n - 1$ elementos que se encontram antes dele.

Os ficheiros diferem dos outros objectos computacionais considerados até aqui ¹Em inglês, “file”.

219

220

CAPÍTULO 8. FICHEIROS

em dois aspectos: (1) os seus valores poderem existir independentemente de qualquer programa, um ficheiro pode existir antes do início da execução de um programa e manter a sua existência após o fim da sua execução; (2) um ficheiro encontra-se, em qualquer instante, num de dois estados possíveis, ou está a ser utilizado para a entrada de dados (estão a ser lidos valores do ficheiro, caso em que se diz que o ficheiro se encontra em modo de leitura) ou está a ser utilizado para a saída de dados (estão a ser escritos valores no ficheiro, caso em que se diz que o ficheiro se encontra em modo de escrita). Ao utilizarmos um ficheiro, temos de dizer ao Python em que modo queremos que esse ficheiro se encontre e qual a localização física dos ficheiros em que os dados se encontram.

8.1

O tipo ficheiro

Sendo os ficheiros entidades que existem fora do nosso programa, antes de utilizar um ficheiro é necessário identificar qual a localização física deste e o modo como o queremos utilizar, ou seja se queremos ler a informação contida no ficheiro ou se queremos escrever informação no ficheiro.

A operação através da qual identificamos a localização do ficheiro e o modo como o queremos utilizar é conhecida por operação de abertura do ficheiro e é realizada em Python recorrendo à função embutida `open`. A função `open` tem a seguinte sintaxe:

`open(expressão, modo, { encoding = tipo })` Esta função tem dois argumentos obrigatórios e um opcional:

- o primeiro argumento, representado por `expressão`, é uma expressão cujo valor é uma cadeia de caracteres que corresponde ao nome externo do ficheiro;
- o segundo argumento, representado por `modo`, é uma expressão cujo valor é uma das cadeias de caracteres `'r'`, `'w'` ou `'a'`. Ou seja,

`modo ::= 'r' 'w' 'a'`

Existem outras alternativas que não são tratadas neste livro.

8.2. LEITURA DE FICHEIROS

221

significando a primeira alternativa que o ficheiro é aberto para leitura³, a segunda alternativa que o ficheiro é aberto para escrita a partir do início do ficheiro⁴ e a terceira alternativa que o ficheiro é aberto para escrita a partir do fim do ficheiro⁵;

- o terceiro argumento, o qual é opcional, é da forma `encoding = tipo`, em que `tipo` é uma expressão que representa o tipo de codificação de caracteres utilizado no ficheiro.

O valor da função `open` corresponde à entidade no programa que está associada ao ficheiro.

8.2

Leitura de ficheiros

Ao abrir um ficheiro para leitura, está subentendido que esse ficheiro existe, pois queremos ler a informação que ele contém. Isto significa que se o Python for instruído para abrir para leitura um ficheiro que não existe, irá gerar um erro como mostra a seguinte interacção:

```
> > > f = open('nada', 'r')
```

`builtins.IOError: [Errno 2] No such file or directory: 'nada'`

Suponhamos que a directoria (ou pasta) do nosso computador que contém os ficheiros utilizados pelo Python continha um ficheiro cujo nome é `teste.txt` e cujo o conteúdo corresponde ao seguinte texto: Este é um teste

que mostra como o Python

lê

e ficheiros de caracteres

A execução pelo Python da instrução

`t = open('teste.txt', 'r', encoding = 'UTF-16')` 3'r é a primeira letra da palavra inglesa “read” (lê).

4'w' é a primeira letra da palavra inglesa “write” (escreve).

5'a' é a primeira letra da palavra inglesa “append” (junta).

222

CAPÍTULO 8. FICHEIROS

E

s

t

e

é

u

m

t

e
s
t
e
\n
q
u
e

Figura 8.1: Representação do ficheiro teste.txt.

cria uma variável no nosso programa cujo nome é `t`, variável essa que está associada ao ficheiro `teste.txt`, o qual pode ser lido pelo Python, sabendo o Python que o texto neste ficheiro está codificado usando o código UTF-16.

Ao ler informação de um ficheiro, o Python mantém um indicador, o indicador de leitura, que indica qual o próximo elemento a ser lido do ficheiro (este indicador é representado nas nossas figuras por uma seta colocada imediatamente por baixo da posição em que se encontra no ficheiro). O indicador de leitura é colocado no início do ficheiro quando o ficheiro é aberto para leitura e movimenta-se no sentido do início para o fim do ficheiro sempre que se efectua uma leitura, sendo colocado imediatamente após o último símbolo lido, de cada vez que a leitura é feita.

Na Figura 8.1 mostramos parte do conteúdo do ficheiro `teste.txt`. Cada um dos elementos deste ficheiro é um carácter, e está representado na figura dentro de um quadrado. Este ficheiro corresponde a uma sequência de caracteres e contém caracteres que não são por nós visíveis quando o inspeccionamos num ecrã e que indicam o fim de cada uma das linhas. Apresentámos na Tabela 2.8

alguns destes caracteres, sendo o fim de linha representado por `\n`. Assim, no ficheiro, imediatamente após a cadeia de caracteres 'Este é um teste', surge o carácter `\n` que corresponde ao fim da primeira linha do ficheiro.

A partir do momento que que é criada uma variável, que designaremos por

❖fich❖, associada a um ficheiro aberto para leitura, passam a existir quatro novas funções no nosso programa para efectuar operações sobre esse ficheiro: 1. ❖fich❖.readline(). Esta função lê a linha do ficheiro ❖fich❖ que se encontra imediatamente a seguir ao indicador de leitura, tendo como valor a cadeia de caracteres correspondente à linha que foi lida. Se o indicador de leitura se encontrar no fim do ficheiro, esta função tem o valor '' (a cadeia de 60s nomes destas funções seguem a sintaxe de ❖nome composto❖ apresentada na página 105, mas na realidade s~

ao um tipo de entidades diferentes. Este aspecto é abordado na Secç~
ao 11.3.

Para os efeitos deste capítulo, iremos considerá-las como simples
funç~

oes.

8.2. LEITURA DE FICHEIROS

223

E

s

t

e

é

u

m

t

e

s

t

```
e
\n
q
u
e
e
```

Figura 8.2: Ficheiro teste.txt após a execução de `t.readline()`.

caracteres vazia). No nosso exemplo, a função que lê uma linha do ficheiro corresponde a `t.readline()`.

2. `fich.readlines()`. Esta função lê todos os caracteres do ficheiro `fich`

que se encontram depois do indicador de leitura, tendo como valor uma lista em que cada um dos elementos é a cadeia de caracteres correspondente a cada uma das linhas que foi lida. Se o indicador de leitura se encontrar no fim do ficheiro, esta função tem o valor `[]` (a lista vazia). No nosso exemplo, esta função corresponde a `t.readlines()`.

3. `fich.read()`. Esta função lê todos os caracteres do ficheiro `fich` que se encontram depois do indicador de leitura, tendo como valor uma cadeia de caracteres contendo todos os caracteres lidos. Se o indicador de leitura se encontrar no fim do ficheiro, esta função tem o valor `''` (a cadeia de caracteres vazia). No nosso exemplo, esta função corresponde a `t.read()`.

4. `fich.close()`. Esta função fecha o ficheiro `fich`. A operação de fecho de um ficheiro corresponde a desfazer a ligação entre o programa e o ficheiro.

No nosso exemplo, a função que fecha o ficheiro corresponde a `t.close()`.

Voltando ao nosso exemplo, suponhamos que abrimos o ficheiro teste.txt com a instrução `t = open('teste.txt', 'r', encoding = 'UTF-16')`. Se executarmos a instrução `ln1 = t.readline()`, a variável `ln1` passa a ter como valor a cadeia de caracteres `'Este é um teste\n'`, ficando o indicador de leitura como se mostra na Figura 8.2. Repare-se

que o carácter de fim de linha faz parte da cadeia de caracteres lida.

A seguinte interacção mostra a utilização da função `t.readline()`. É importante notar que ao atingir o fim do ficheiro, a função `t.readline()` tem como valor a cadeia de caracteres vazia, `''`. Este aspecto é importante quando o nosso programa está a ler um ficheiro e necessita de determinar quando a leitura chega ao fim.

224

CAPÍTULO 8. FICHEIROS

```
> > > t = open('teste.txt', 'r', encoding = 'UTF-16')
```

```
> > > ln1 = t.readline()
```

```
> > > ln1
```

```
'Este é um teste\n'
```

```
> > > ln2 = t.readline()
```

```
> > > ln2
```

```
'que mostra como o Python\n'
```

```
> > > ln3 = t.readline()
```

```
> > > ln3
```

```
'\n'
```

```
e ficheiros de caracteres\n'
```

```
> > > ln4 = t.readline()
```

```
> > > ln4
```

```
''
```

```
> > > t.close()
```

A seguinte interacção mostra a utilização das funções disponíveis para ler ficheiros, indicando o comportamento de cada uma delas ao serem avaliadas com situações diferentes relativas ao indicador de leitura.

```
> > > f = open('teste.txt', 'r', encoding = 'UTF-16')
```

```
> > > l1 = f.readline()
```

```
> > > l1
```

```
'Este é um teste\n'
```

```
> > > l2 = f.read()
```

```
> > > l2
```

```
'que mostra como o Python\nl'
```

```
e ficheiros de caracteres\n'
```

```
> > > print(l2)
```

```
que mostra como o Python
```

```
l'
```

```
e ficheiros de caracteres
```

```
> > > f.close()
```

```
> > > g = open('teste.txt', 'r', encoding = 'UTF-16')
```

```
> > > lines = g.readlines()
```

```
> > > lines
```

```
['Este é um teste\n', 'que mostra como o Python\n',
```

```
l'
```

```
e ficheiros de caracteres\n']
```

8.2. LEITURA DE FICHEIROS

225

Vamos agora analisar de uma forma mais detalhada os argumentos da função `open`. O nome do ficheiro utilizado por esta função corresponde à especificação completa do nome do ficheiro a utilizar. No exemplo que apresentámos, o ficheiro `teste.txt` existia na directoria (ou pasta) do nosso computador que é utilizada por omissão pelo Python. Contudo, se este ficheiro não existisse na directoria de omissão, mas sim numa directoria chamada `exemplos` localizada na directoria de omissão do Python, o nome do ficheiro a especificar na

função ao open seria

'exemplos/teste.txt'. Ou seja, o primeiro argumento da função open não é o nome de um ficheiro, mas sim a combinação de um caminho de directorias e do nome de um ficheiro. Embora diferentes sistemas operativos usem símbolos diferentes para especificar caminhos em directorias (o Mac OS e o Linux usam o símbolo `,` ao passo que o Windows usa o símbolo `\`), na função open é sempre utilizado o símbolo

O argumento da função open associado à codificação de caracteres é ligeiramente mais complicado. Sabemos que os caracteres correspondem a símbolos e que estes são representados internamente em Python utilizando o Unicode (este aspecto foi discutido na Secção 4.3). Internamente ao Python, uma cadeia de caracteres é uma sequência de zeros e uns correspondente a representações de caracteres usando o Unicode. Contudo, um ficheiro existente no disco não é uma sequência de símbolos codificados em Unicode, mas apenas uma sequência de zeros e uns. Ao ler um ficheiro de texto existente num disco, o Python precisa de saber como “interpretar” a sequência de zeros e uns nele contida. Com base nesta informação, o Python descodifica a sequência de zeros e uns existente no disco, devolvendo uma sequência de caracteres em Unicode, que é identificada como uma cadeia de caracteres.

Para tornar as coisas ainda mais complicadas, não só a codificação de informação em disco é dependente do tipo de computador, mas também, dentro do mesmo computador existem normalmente ficheiros com diferentes tipos de codificação.

Os sistemas operativos lidam com esta diversidade de codificações porque cada ficheiro contém informação sobre a codificação que este utiliza, informação essa que não está acessível ao Python. Por estas razões, ao abrir um ficheiro, é necessário dizer ao Python qual o tipo de codificação utilizada.

226

CAPÍTULO 8. FICHEIROS

8.3

Escrita em ficheiros

De modo a escrevermos informação num ficheiro, teremos primeiro que efectuar a abertura do ficheiro com um dos modos 'w' ou 'a'. Ao

abrir um ficheiro para escrita, se o ficheiro não existir, ele é criado pelo Python como um ficheiro sem elementos. Tal como no caso da leitura em ficheiros, ao escrever informação num ficheiro, o Python mantém um indicador, o indicador de escrita, que indica qual a posição do próximo elemento a ser escrito no ficheiro. Consoante o modo escolhido para a abertura do ficheiro, o Python coloca o indicador de escrita ou no início do ficheiro (ou seja, o ficheiro fica sem quaisquer elementos, e o seu antigo conteúdo, se existir, é perdido), se for utilizado o modo 'w', ou no fim do ficheiro, se for utilizado o modo 'a'.

De um modo semelhante ao que acontece quando abrimos um ficheiro em modo de leitura, a partir do momento que é criada uma variável, que designaremos por `fich`, associada a um ficheiro aberto para escrita, passam a existir as seguintes funções no nosso programa para efectuar operações sobre esse ficheiro: 1.

`fich.write(cadeia de caracteres)`. Esta função escreve, a partir da posição do indicador de escrita, a cadeia de caracteres no ficheiro `fich`.

O indicador de escrita é movimentado para a posição imediatamente a seguir à cadeia de caracteres escrita. Esta função devolve o número de caracteres escritos no ficheiro.

2. `fich.writelines(sequência)`, na qual `sequência` é um tuplo ou uma lista cujos elementos são cadeias de caracteres. Esta função escreve, a partir da posição do indicador de escrita, cada um dos elementos da

`sequência` no ficheiro `fich`, não escrevendo o carácter de fim de linha entre os elementos escritos. O indicador de escrita é movimentado para a posição imediatamente a seguir à última cadeia de caracteres escrita. Esta função não devolve nenhum valor.

3. `fich.close()`. Esta função fecha o ficheiro `fich`.

A seguinte interacção mostra a utilização de operações de escrita e de leitura num ficheiro. Poderá parecer estranho a não utilização da indicação sobre a codificação dos caracteres utilizada no ficheiro. Contudo, como os ficheiros utilizados são criados pelo Python, a não indicação da codificação significa que esta será a codificação usada por omissão pelo Python.

8.3. ESCRITA EM FICHEIROS


```
> > > f1 = open('teste1', 'w')
```

```
> > > f1.write('abc')
```

```
3
```

```
> > > f1.write('def')
```

```
3
```

```
> > > f1.close()
```

```
> > > f1 = open('teste1', 'r') # 'teste1' é aberto para leitura
```

```
> > > cont = f1.read()
```

```
> > > print(cont)
```

```
> > > abcdef
```

```
> > > cont # inspecç~
```

ao do conteúdo do ficheiro f1

```
'abcdef'
```

```
> > > f1.close()
```

```
> > > # 'teste1' (já existente) é aberto para escrita
```

```
> > > f1 = open('teste1', 'w')
```

```
> > > f1.close()
```

```
> > > f1 = open('teste1', 'r') # 'teste1' é aberto para leitura
```

```
> > > cont = f1.read()
```

```
> > > cont # o conteúdo do ficheiro foi apagado
```

```
”
```

```
> > > f1.close()
```

A interacç~ao anterior mostra que se um ficheiro existente é aberto para leitura, o seu conteúdo é apagado com a operaç~ao de abertura do ficheiro.

Após a abertura de um ficheiro para escrita, é também possível utilizar a função `print` apresentada na Secção 2.5.2, a qual, na realidade, tem uma sintaxe definida pelas seguintes expressões em notação BNF:

`escrita de dados ::= print()`

`print(file = nome de ficheiro)`

`print(expressões)`

`print(expressões, file = nome de ficheiro)`

`nome de ficheiro ::= nome`

Para as duas novas alternativas aqui apresentadas, a semântica desta função é definida do seguinte modo: ao encontrar a invocação da função `print(file =`

`nome)`, o Python escreve uma linha em branco no ficheiro `nome`; ao encontrar

228

CAPÍTULO 8. FICHEIROS

a invocação da função `print(exp1, ... expn, file = nome)`, o Python avalia cada uma das expressões `exp1 ... expn`, escrevendo-as na mesma linha do ficheiro `nome`, separadas por um espaço em branco e terminando com um salto de linha.

A seguinte interacção mostra a utilização da função `print` utilizando nomes de ficheiros:

```
>>> f1 = open('teste1', 'w')
```

```
# 'teste1' é aberto para escrita
```

```
>>> f1.write('abc')
```

```
3
```

```
>>> print('cde', file = f1)
```

```
>>> print('fgh', 5, 5 * 5, file = f1)
```

```
> > > print('ijk', file = f1)

> > > f1.close()

> > > f1 = open('teste1', 'r')

# 'teste1' é aberto para leitura

> > > cont = f1.read()

> > > cont

# conteúdo de 'teste1'

'abccde\nfgh 5 25\nijk\n'

> > > print(cont)

print('abccde\nfgh 5 25\nijk\n')

abccde

fgh 5 25

ijk

> > > f1.close()

> > > f1 = open('teste1', 'a') # 'teste1' é aberto para adiç~

ao

> > > print(file = f1)

> > > print(file = f1)

> > > f1.write('lmn')

3

> > > f1.close()

> > > f1 = open('teste1', 'r')

# 'teste1' é aberto para leitura

> > > cont = f1.read()
```

```
> > > cont
```

```
# conteúdo de 'teste1'
```

```
'abccde\nfgh 5 25\nijk\n\n\nlmn'
```

```
> > > print(cont)
```

8.4. NOTAS FINAIS

229

abccde

fgh 5 25

ijk

lmn

```
> > > f1.close()
```

8.4

Notas finais

Apresentámos o conceito de ficheiro, considerando apenas ficheiros de texto.

Um ficheiro corresponde a uma entidade que existe no computador independentemente da execução de um programa.

Para utilizar um ficheiro é necessário abrir o ficheiro, ou seja, associar uma entidade do nosso programa com um ficheiro físico existente no computador ou na rede a que o computador está ligado e dizer qual o tipo de operações a efectuar no ficheiro, a leitura ou a escrita de informação.

Depois de aberto um ficheiro apenas é possível efectuar operações de leitura ou de escrita, dependendo do modo como este foi aberto.

8.5

Exercícios

1. Escreva um programa em Python que lê informação de dois ficheiros, cada um dos quais contém números ordenados por ordem

crescente, contendo cada linha dos ficheiros apenas um número. O seu programa produz um ficheiro ordenado de números (contendo um número por linha) correspondente à junção dos números existentes nos dois ficheiros. Para cada um dos ficheiros de entrada, o seu programa só pode ler uma linha de cada vez.

2. Escreva um programa em Python para formatar texto. O seu programa deve pedir ao utilizador o nome do ficheiro que contém o texto a ser formatado (este ficheiro, designado por ficheiro fonte, contém o texto pro-priamente dito e os comandos para o formador de texto tal como se

230

CAPÍTULO 8. FICHEIROS

descrevem a seguir) e o nome do ficheiro onde o texto formatado será guardado (o ficheiro de destino). Após esta interacção deve ser iniciado o processamento do texto, utilizando os seguintes valores de omissão:

- Cada página tem espaço para 66 linhas;
- O número de linhas de texto em cada página é de 58;
- A margem esquerda começa na coluna 9;
- A margem direita acaba na coluna 79;
- As linhas são geradas com espaçamento simples (isto é, não são inseridas linhas em branco entre as linhas de texto);
- As páginas são numeradas automaticamente, sendo o número da página colocado no canto superior direito;
- Os parágrafos (indicados no ficheiro fonte por uma linha que começa com um ou mais espaços em branco) são iniciados 8 espaços mais para a direita e separados da linha anterior por uma linha em branco;
- Cada linha do texto começa na margem da esquerda e nunca ultrapassa a margem da direita.

Estes valores de omissão podem ser alterados através de comandos fornecidos ao programa, comandos esses que são inseridos em qualquer ponto do texto. Cada comando é escrito numa linha individualizada que contém o carácter “\$” na primeira coluna. Os

comandos possíveis são: \$ nl

Este comando faz com que o texto comece numa nova linha, ou seja, origina uma nova linha no ficheiro de destino.

\$ es $\diamond n \diamond$

Este comando faz com que o espaçamento entre as linhas do texto passe a ser de $\diamond n \diamond$ linhas em branco (o símbolo não terminal $\diamond n \diamond$ corresponde a um inteiro positivo).

\$ sp $\diamond n \diamond$

Este comando insere $\diamond n \diamond$ linhas em branco no texto, tendo em atenção o valor do espaçamento (o símbolo não terminal $\diamond n \diamond$ corresponde a um inteiro positivo).

8.5. EXERCÍCIOS

231

\$ ce $\diamond \text{texto} \diamond$

Este comando causa um salto de linha e centra, na linha seguinte, a cadeia de caracteres representada pelo símbolo não terminal $\diamond \text{texto} \diamond$.

\$ al

Depois da execução deste comando, todas as linhas produzidas pelo programa estão alinhadas, ou seja, as linhas começam na margem esquerda e acabam na margem direita, embora o número de caracteres possa variar de linha para linha.

Para conseguir este comportamento, o seu programa insere espaços adicionais entre as palavras da linha de modo a que esta termine na margem direita.

\$ na

Depois da execução deste comando, as linhas produzidas pelo programa não têm de estar alinhadas, ou seja, as linhas começam na margem da esquerda e podem acabar antes da margem da direita, desde que a palavra seguinte não caiba na presente linha. Neste caso, não se verifica a inserção de espaços adicionais entre palavras.

\$ me $\diamond n \diamond$

Depois da execu~ao deste comando, a margem da esquerda passa a comecar na coluna correspondente ao s'imbolo n~ao terminal $\diamond n \diamond$. Se o valor de $\diamond n \diamond$ for maior ou igual ao valor da margem da direita, é originado um erro de execu~ao e este comando é ignorado (o s'imbolo n~ao terminal $\diamond n \diamond$ corresponde a um inteiro positivo).

\$ md $\diamond n \diamond$

Depois da execu~ao deste comando, a margem da direita passa a acabar na coluna correspondente ao s'imbolo n~ao terminal $\diamond n \diamond$. Se o valor de

$\diamond n \diamond$ for menor ou igual ao valor da margem da esquerda, é originado um erro de execu~ao e este comando é ignorado (o s'imbolo n~ao terminal

$\diamond n \diamond$ corresponde a um inteiro positivo).

\$ pa

A execu~ao deste comando causa um salto de página, excepto se, quando ele for executado, o formatador de texto se encontrar no início de uma página.

Sugest~ao: Utilize uma cadeia de caracteres para armazenar a linha que está a ser gerada para o ficheiro de destino. O seu programa deve ler

232

CAPÍTULO 8. FICHEIROS

palavras do ficheiro fonte e decidir se estas cabem ou não na linha a ser gerada. Em caso afirmativo, estas são adicionadas à linha a ser gerada, em caso contrário a linha é escrita no ficheiro de destino, eventualmente depois de algum processamento.

Capítulo 9

Dicionários

They were standing under a tree, each with an arm round the other's neck, and Alice knew which was which in a moment, because one of them had 'DUM' embroidered on his collar, and the other 'DEE.'

'I suppose they've each got "TWEEDLE" round at the back of the collar,' she said to herself.

9.1

O tipo dicionário

Um dicionário, em Python designado por `dict`¹, em programação normalmente conhecido como lista associativa ou tabela de dispersão², é um tipo mutável contendo um conjunto de pares. O primeiro elemento de cada par tem o nome de chave e o segundo elemento do par corresponde ao valor associado com a chave.

Num dicionário não podem existir dois pares distintos com a mesma chave.

Esta definição de dicionário é semelhante à definição de função apresentada na página 72, contudo, a utilização que faremos dos dicionários corresponde à manipulação directa do conjunto de pares que caracteriza um dado dicionário (recorrendo à definição por extensão), em lugar da manipulação da expressão designatória que define esse dicionário.

Notemos também, que ao passo que as duas estruturas de informação que consi-

Do inglês, “hash table”.

235

236

CAPÍTULO 9. DICIONÁRIOS

derámos até agora, os tuplos e as listas, correspondem a sequências de elementos, significando isto que a ordem pela qual os elementos surgem na sequência é importante, os dicionários correspondem a conjuntos de elementos, o que significa que a ordem dos elementos num dicionário é irrelevante.

Utilizando a notação BNF, a representação externa de dicionários em Python é definida do seguinte modo:

◆dicionário◆ ::= { }

{ ◆pares◆ }

◆pares◆ ::= ◆par◆

❖par❖, ❖pares❖

❖par❖ ::= ❖chave❖ : ❖valor❖

❖chave❖ ::= ❖express~ao❖

❖valor❖ ::= ❖express~ao❖

❖tuplo❖

❖lista❖

❖dicion~ario❖

Os elementos de um dicion~ario s~ao representados dentro de chavetas, podendo corresponder a qualquer n~umero de elementos, incluindo zero. O dicion~ario {}

tem o nome de dicion~ario vazio.

O Python exige que a chave seja um elemento de um tipo imut~avel, tal como um n~umero, uma cadeia de caracteres ou um tuplo. Esta restriç~ao n~ao pode ser representada em notaç~ao BNF. O Python n~ao imp~oe restriç~oes ao valor associado com uma chave.

S~ao exemplos de dicion~arios, {'a':3, 'b':2, 'c':4}, {chr(56 + 1):12} (este dicion~ario é o mesmo que {'9':12}), {(1, 2):'amarelo', (2, 1):'azul'} e

{1:'c', 'b':4}, este ~ultimo exemplo mostra que as chaves de um dicion~ario podem ser de tipos diferentes. A entidade {'a':3, 'b':2, 'a':4} n~ao é um dicion~ario pois tem dois pares com a mesma chave 'a'.3

É importante recordar que os dicion~arios s~ao conjuntos, pelo que quando o Python mostra um dicion~ario ao utilizador, os seus elementos podem n~ao aparecer pela mesma ordem pela qual foram escritos, como o demonstra a seguinte 3Na realidade, se fornecermos ao Python a entidade {'a':3, 'b':2, 'a':4}, o Python considera-a como o dicion~ario {'b':2, 'a':4}. Este aspecto está relacionado com a representaç~

ao interna de dicion~arios, a qual pode ser consultada em [Cormen et al., 2009].

9.1. O TIPO DICIONÁRIO

interacção:

```
> > > {'a':3, 'b':2, 'c':4}
```

```
{'a': 3, 'c': 4, 'b': 2}
```

Os elementos de um dicionário são referenciados através do conceito de nome indexado apresentado na página 113. Em oposição aos tuplos e às listas que, sendo sequências, os seus elementos são acedidos por um índice que correspondem à sua posição, os elementos de um dicionário são acedidos utilizando a sua chave como índice. Assim, sendo *d* um dicionário e *c* uma chave do dicionário *d*, o nome indexado *d*[*c*] corresponde ao valor associado à chave *c*. O

acesso e a modificação dos elementos de um dicionário são ilustrados na seguinte interacção:

```
> > > ex_dic = {'a':3, 'b':2, 'c':4}
```

```
> > > ex_dic['a']
```

```
3
```

```
> > > ex_dic['d'] = 1
```

```
> > > ex_dic
```

```
{'a': 3, 'c': 4, 'b': 2, 'd': 1}
```

```
> > > ex_dic['a'] = ex_dic['a'] + 1
```

```
> > > ex_dic['a']
```

```
4
```

```
> > > ex_dic['j']
```

```
KeyError: 'j'
```

Na segunda linha, `ex_dic['a']` corresponde ao valor do dicionário associado à chave 'a'. Na quarta linha, é definido um novo par no dicionário `ex_dic`, cuja chave é 'd' e cujo valor associado é 1. Na sétima linha, o valor associado à chave 'a' é aumentado em uma unidade. As últimas duas linhas mostram que se uma chave não existir num dicionário, uma referência a essa chave dá origem a um erro.

Sobre os dicionários podemos utilizar as funções embutidas apresentadas na Tabela 9.1 como se ilustra na interação seguinte.

```
> > > ex_dic
```

```
{'a': 4, 'c': 4, 'b': 2, 'd': 1}
```

238

CAPÍTULO 9. DICIONÁRIOS

Operação

Tipo dos

Valor

argumentos

c in d

Chave

True se a chave c pertence ao dicionário d;

e dicionário False em caso contrário.

c not in d

Chave

A negação do resultado da operação c in d.

e dicionário

dict()

—

O dicionário vazio.

len(d)

Dicionário

O número de elementos do dicionário d.

Tabela 9.1: Operações sobre dicionários em Python.

```
> > > 'f' in ex_dic
```

False

```
> > > 'a' in ex_dic
```

True

```
> > > len(ex_dic)
```

4

Analogamente ao que acontece com tuplos e listas, podemos utilizar um ciclo for para percorrer todos os elementos de um dicionário, como o mostra a seguinte interacção:

```
> > > inventores = {'Dean Kamen': 'Segway',
```

```
... 'Tim Berners-Lee': 'World Wide Web',
```

```
... 'Dan Bricklin': 'Spreadsheet'}
```

```
...
```

```
> > > for i in inventores:
```

```
...
```

```
print(i, ' inventou: ', inventores[i])
```

```
...
```

Dean Kamen

inventou:

Segway

Tim Berners-Lee

inventou:

World Wide Web

Dan Bricklin

inventou:

9.2

Frequência de letras num texto

Suponhamos que Lusiadas.txt é um ficheiro de texto, contendo o texto completo de Os Lusíadas. O seguinte programa lê a totalidade do ficheiro Lusia-

9.2. FREQUÊNCIA DE LETRAS NUM TEXTO

239

das.txt, criando um dicionário em que as chaves são caracteres e cada carácter está associado ao número de vezes que este aparece no texto de Os Lusíadas.

```
lus = open('Lusiadas.txt', 'r', encoding = 'UTF-16') cont = {}
```

```
linha = lus.readline()
```

```
while linha != "": # enquanto o fim do ficheiro não
```

```
ao for atingido
```

```
for c in linha: # o dicionário é actualizado if c not in cont:
```

```
cont[c] = 1
```

```
else:
```

```
cont[c] = cont[c] + 1
```

```
linha = lus.readline()
```

```
lus.close()
```

Note-se que neste programa, o dicionário criado inclui uma chave associada ao carácter de salto de linha (\n), ao espaço em branco e a todos os símbolos de pontuação. Após a execução deste programa podemos gerar a interacção:

```
> > > cont['a']
```

29833

```
> > > cont['z']
```

913

```
> > > cont['\n']
```

9942

```
> > > cont[' ']
```

50449

```
> > > cont['.']
```

1618

Se desejarmos ignorar os saltos de linha, os espaços em branco e os símbolos de pontuação, podemos criar uma lista chamada `pontuacao` que contém os caracteres a ignorar⁴:

```
pontuacao = ['!', '"', "'", '(', ')', '-', ',', '.', \
',', ':', '?', '[', ']', '{', '}', ' ', '\n']
```

⁴Note-se a utilização de dois delimitadores para cadeias de caracteres, uma contendo as aspas e a outra a plica.

240

CAPÍTULO 9. Dicionários

Usando esta lista, podemos alterar o ciclo `for` do pelo programa do seguinte modo:

```
for c in linha:
```

```
if not c in pontuacao:
```

```
if c not in cont :
```

```
cont[c] = 1
```

```
else:
```

```
cont[c] = cont[c] + 1
```

No entanto, o nosso programa ainda conta separadamente as letras minúsculas e as letra maiúsculas que aparecem no texto, como o mostra a seguinte interação:

```
> > > cont['a']
```

29833

```
> > > cont['A']
```

1222

Para resolver este problema podemos pensar em escrever uma função que transforma letras maiúsculas em minúsculas. Esta função recorre ao Unicode, calculando qual a letra minúscula correspondente a uma dada maiúscula. Para as letras não acentuadas, esta função é simples de escrever; para símbolos acentuados, esta função obriga-nos a determinar as representações de cada letra maiúscula acentuada, por exemplo “Á” e a sua minúscula correspondente “á”.

De modo a que o nosso programa seja completo, teremos também que fazer o mesmo para letras acentuadas noutras línguas que não o português, por exemplo

“Ü” e “ü”. Uma outra alternativa, a que vamos recorrer, corresponde a utilizar uma função embutida do Python, a qual para uma dada cadeia de caracteres, representada pela variável *s*, transforma essa cadeia de caracteres numa cadeia equivalente em que todos os caracteres correspondem a letras minúsculas. Esta função, sem argumentos tem o nome *s.lower()*⁵. Por exemplo:

```
> > > ex_cadeia = 'Exemplo DE CONVERS~
```

```
AO'
```

```
> > > ex_cadeia.lower()
```

```
'exemplo de convers~
```

```
ao'
```

5 Novamente, surge-nos aqui uma situação semelhante às funções de leitura e escrita em ficheiros, que que o nome da função

ao é um  nome composto  em que um dos seus constituintes é o nome de uma variável. Este aspecto é abordado na Secção

ao 11.3.

9.2. FREQUÊNCIA DE LETRAS NUM TEXTO

Podemos agora apresentar a seguinte função que conta o número de letras existentes num ficheiro de texto, cujo nome `lhe` é fornecido como argumento: `def conta_letras (nome):`

```
fich = open(nome, 'r', encoding = 'UTF-16')

cont = {}

pontuacao = ['!', '"', "'", '(', ')', '-', ',', '.', \
',', ':', '?', '[', ']', '{', '}', ' ', '\n']

linha = fich.readline()

while linha != "": # enquanto o fim do ficheiro não
    ao for atingido

    for c in linha.lower(): # transformaç~
        ao em minúsculas

        if not c in pontuacao:

            if c not in cont:

                cont[c] = 1

            else:

                cont[c] = cont[c] + 1

    linha = fich.readline()

    fich.close()

    return cont
```

Obtendo a interação:

```
> > > cont = conta_palavras('Lusiadas.txt')

> > > cont
```


e': 378, 'ʹ

i': 520, '˘

a': 1619, 'p': 5554, 'ó': 660, 'ú': 163,

'ò': 1, 'u': 10617, 'è': 2, 'ˆ

o': 52, 'à': 1377, 'ç': 863, 'ˆ

a': 79,

'a': 31055, 'à': 158, 'c': 7070, 'b': 2392, 'e': 31581,

'd': 12304, 'g': 3598, 'f': 3055, 'é': 823, 'h': 2583, 'j': 1023,

'l': 6098, 'o': 27240, 'n': 13449, 'q': 4110, 'i': 12542,

's': 20644, 'r': 16827, '˘

o': 115, 't': 11929, 'v': 4259, 'y': 7,

'x': 370, 'z': 919, 'ü': 14,

'm': 10912}

Suponhamos que após a execução do programa anterior, executávamos as seguintes instruções:

```
total_letras = 0
```

```
for c in cont:
```

```
total_letras = total_letras + cont[c]
```

242

CAPÍTULO 9. DICIONÁRIOS

Com estas instruções calculamos o número total de letras no texto de Os Lusíadas, a partir do qual podemos calcular a frequência relativa de cada letra: $\text{freq_rel} = \{\}$

```
for c in cont:
```

```
freq_rel[c] = cont[c] / total_letras
```

Após a execução deste programa podemos gerar a interação:

```
> > > total_letras
```

```
246962
```

```
> > > freq_rel['a']
```

```
0.12574809079939425
```

```
> > > freq_rel['z']
```

```
0.003721220268705307
```

```
9.3
```

Dicionários de dicionários

Suponhamos que desejávamos representar informação relativa à ficha académica de um aluno numa universidade, informação essa que deverá conter o número do aluno, o seu nome e o registo das disciplinas que frequentou, contendo para cada disciplina o ano lectivo em que o aluno esteve inscrito e a classificação obtida.

Comecemos por pensar em como representar a informação sobre as disciplinas frequentadas. Esta informação corresponde a uma colecção de entidades que associam o nome (ou a abreviatura) da disciplina às notas obtidas e o ano lectivo em que a nota foi obtida. Para isso, iremos utilizar um dicionário em que as chaves correspondem às abreviaturas das disciplinas e o seu valor ao registo das notas realizadas. Para um aluno que tenha frequentado as disciplinas de FP, AL, TC e SD, este dicionário terá a forma (para facilidade de leitura, escrevemos cada par numa linha separada):

```
{'FP': ♦registo das classificações obtidas em FP♦,
```

```
'AL': ♦registo das classificações obtidas em AL♦,
```

```
'TC': ♦registo das classificações obtidas em TC♦,
```

```
'SD': ♦registo das classificações obtidas em SD♦}
```

9.3. DICIONÁRIOS DE DICIONÁRIOS

```
243
```

Consideremos agora a representação do registo das classificações obtidas numa dada disciplina. Um aluno pode frequentar uma disciplina mais do que uma vez.

No caso de reprovaç~ao na disciplina, frequentará a disciplina até obter aprovaç~ao (ou prescrever); no caso de obter aprovaç~ao, poderá frequentar a disciplina para melhoria de nota no ano lectivo seguinte. Para representar o registo das classificaç~oes obtidas numa dada disciplina, iremos utilizar um dicionário em que cada par contém o ano lectivo em que a disciplina foi frequentada (uma cadeia de caracteres) e o valor corresponde à classificaç~ao obtida, um inteiro entre 10 e 20 ou REP. Ou seja, o valor associado a cada disciplina é por sua vez um dicionário! O seguinte dicionário pode corresponder às notas do aluno em consideraç~ao:

```
{'FP': {'2010/11': 12, '2011/12': 15},  
'AL': {'2010/11': 10},  
'TC': {'2010/11': 12},  
'SD': {'2010/11': 'REP', '2011/12': 13}}
```

Se o dicionário anterior tiver o nome `disc`, ent~ao podemos gerar a seguinte interacç~ao:

```
> > > disc['FP']  
{'2010/11': 12, '2011/12': 15}  
  
> > > disc['FP']['2010/11']
```

12

O nome de um aluno é constituído por dois componentes, o nome próprio e o apelido. Assim, o nome de um aluno será também representado por um dicionário com duas chaves, 'nome p' e 'apelido', correspondentes, respectivamente ao nome próprio e ao apelido do aluno. Assim, o nome do aluno António Mega Bites é representado por:

```
{'nome_p': 'António', 'apelido': 'Mega Bites'}
```

Representaremos a informaç~ao sobre um aluno como um par pertencente a um dicionário cujas chaves correspondem aos números dos alunos e cujo valor associado é um outro dicionário com uma chave 'nome' à qual está associada o nome do aluno e com uma chave 'disc', à qual está associada um dicionário com as classificaç~oes obtidas pelo aluno nos diferentes anos lectivos.

CAPÍTULO 9. DICIONÁRIOS

Por exemplo, se o aluno N.º. 12345, com o nome António Mega Bites, obteve as classificações indicadas acima, a sua entrada no dicionário será representada por (para facilidade de leitura, escrevemos cada par numa linha separada):

```
{12345: {'nome': {'nome_p': 'António', 'apelido': 'Mega Bites'},
'disc': {'FP': {'2010/11': 12, '2011/12': 15},
'AL': {'2010/11': 10},
'TC': {'2010/11': 12},
'SD': {'2010/11': 'REP', '2011/12': 13}}}}
```

Sendo alunos o dicionário com informação sobre todos os alunos da universidade, podemos obter a seguinte interacção:

```
> > > alunos[12345]['nome']
{'nome_p': 'António', 'apelido': 'Mega Bites'}
> > > alunos[12345]['nome']['apelido']
'Mega Bites'
> > > alunos[12345]['disc']['FP']['2010/11']
```

12

Vamos agora desenvolver uma função utilizada pela secretaria dos registos académicos ao lançar as notas de uma dada disciplina. O professor responsável da disciplina envia à secretaria por correio electrónico um ficheiro de texto contendo na primeira linha a identificação da disciplina, na segunda linha o ano lectivo ao qual as notas correspondem, seguido de uma série de linhas, cada uma contendo o número de um aluno e a nota respectiva. Por exemplo, o seguinte ficheiro poderá corresponder às notas obtidas em FP no ano lectivo de 2012/13: FP

2012/13

12345 12

12346 REP

12347 10

12348 14

12349 REP

12350 16

9.3. DICIONÁRIOS DE DICIONÁRIOS

245

A função `introduz_notas` recebe o ficheiro contendo as notas de uma disciplina e o dicionário contendo a informação sobre os alunos e efectua o lançamento das notas de uma determinada disciplina, num dado ano lectivo. O dicionário contendo a informação sobre os alunos é passado por referência, pelo que qualquer alteração a este dicionário se reflecte no correspondente parâmetro concreto.

```
def introduz_notas (origem, registos):
```

```
    f = open(origem, 'r')
```

```
    disc = f.readline()
```

```
    disc = disc[0:len(disc) - 1] # remove fim de linha
    ano_lectivo = f.readline()
```

```
    ano_lectivo = ano_lectivo[0:len(ano_lectivo) - 1] # idem
    info_aluno = f.readline()
```

```
    while info_aluno != "":
```

```
        num, nota = obtem_info(info_aluno)
```

```
        if num not in registos:
```

```
            print('O aluno ', num, ' não
```

```
            existe')
```

```
        else:
```

```
            registos[num]['disc'][disc][ano_lectivo] = nota
            info_aluno = f.readline()
```

```
    f.close()
```

A função `obtem_info` recebe uma linha (uma cadeia de caracteres) correspondente a um número de aluno e a respectiva nota e devolve um tuplo contendo o número do aluno como um inteiro e a nota

correspondente.

```
def obtem_info(linha):
```

```
    pos = 0;
```

```
    num = ''
```

```
    nota = ''
```

```
    while linha[pos] != ' ': # obtém número
```

```
        num = num + linha[pos]
```

```
    pos = pos + 1
```

```
    num = eval(num)
```

```
    while linha[pos] == ' ': # salta sobre brancos pos = pos + 1
```

```
246
```

CAPÍTULO 9. DICIONÁRIOS

```
while linha[pos] != '\n': # obtém nota
```

```
    nota = nota + linha[pos]
```

```
    pos = pos + 1
```

```
if nota != 'REP':
```

```
    nota = eval(nota)
```

```
    return (num, nota)
```

9.4

Caminhos mais curtos em grafos

Como último exemplo da utilização de dicionários, apresentamos um algoritmo para calcular o caminho mais curto entre os nós de um grafo. Um grafo corresponde a uma descrição formal de um grande número de situações do mundo real. Um dos exemplos mais comuns da utilização de grafos corresponde à representação de um mapa com ligações entre cidades. Podemos considerar um mapa como sendo constituído por um conjunto de nós, os quais correspondem às cidades do mapa, e por um conjunto de arcos, os quais correspondem

às ligações entre as cidades. Formalmente, um grafo corresponde a uma estrutura (N, A) , na qual N é um conjunto de entidades a que se dá o nome de nós e A é um conjunto de ligações, os arcos, entre os nós do grafo. Os arcos de um grafo podem ser dirigidos, correspondentes, na analogia com os mapas, a estradas apenas com um sentido, e podem também ser rotulados, contendo, por exemplo, a distância entre duas cidades seguindo uma dada estrada ou o valor da portagem a pagar se a estrada correspondente for seguida.

Na Figura 9.1, apresentamos um grafo com seis nós, com os identificadores A, B, C, D, E e F. Neste grafo existe, por exemplo, um arco de A para B, com a distância 10, um arco de C para D, com a distância 15 e um arco de D para C, também com a distância 15. Neste grafo, por exemplo, não existe um arco do nó A para o nó F. Dado um grafo e dois nós desse grafo, n_1 e n_2 , define-se um caminho de n_1 e n_2 como a sequência de arcos que é necessário atravessar para ir do nó n_1 para o nó n_2 . Por exemplo, no grafo da Figura 9.1, um caminho de A para D corresponde ao arcos que ligam os nós A a B, B a C e C a D.

Uma aplicação comum em grafos corresponde a determinar a distância mais curta entre dois nós, ou seja, qual o caminho entre os dois nós em que o somatório das distâncias percorridas é menor. Um algoritmo clássico para determinar a distância mais curta entre dois nós de um grafo foi desenvolvido por Edsger

9.4. CAMINHOS MAIS CURTOS EM GRAFOS

247

F

15

25

5

20

A

10

B

15

E
5
8
15
C
D
15

Figura 9.1: Exemplo de um grafo.

Dijkstra⁶ (1930–2002).

O algoritmo de Dijkstra recebe como argumentos um grafo e um nó desse grafo, a que se chama o nó inicial, e calcula a distância mínima do nó inicial a cada nó do grafo. O algoritmo começa por associar cada nó com a distância conhecida ao nó inicial. A esta associação chamamos a distância calculada. No início do algoritmo, apenas sabemos que a distância do nó inicial a si próprio é zero, podendo a distância do nó inicial a qualquer outro nó ser infinita (se não existirem ligações entre eles). Por exemplo, em relação ao grafo apresentado na Figura 9.1, sendo A o nó inicial, representaremos as distâncias calculadas iniciais por:

{'A':0, 'B':inf, 'C':inf, 'D':inf,
'E':inf,
'F':inf}

em que inf representa infinito. Na Figura 9.2 (a), apresentamos dentro de parênteses, junto a cada nó, a distância calculada pelo algoritmo. No início do algoritmo cria-se também uma lista contendo todos os nós do grafo.

O algoritmo percorre repetitivamente a lista dos nós, enquanto esta não for vazia, executando as seguintes acções:

1. Escolhe-se da lista de nós o nó com menor valor da distância calculada.

Se existir mais do que um nó nesta situação, escolhe-se

arbitrariamente um deles;

2. Remove-se o nó escolhido da lista de nós; 6[Dijkstra, 1959].

248

CAPÍTULO 9. DICIONÁRIOS

(inf)

(inf)

F

F

15

25

25

5

20

15

5

20

(inf)

(10)

A

10

B

15

E

(inf)

A

10

B

15

E

(inf)

(0)

(0)

5

8

5

8

15

15

C

D

C

D

15

15

(inf)

(inf)

(inf)

(inf)

(a)

(b)

Figura 9.2: Dois primeiros passos no algoritmo de Dijkstra.

(15)

(15)

F

F

15

25

25

5

20

15

5

20

(10)

(15)

A

10

B

15

E

(25)

A

10

B

15

E

(25)

(0)

(0)

5

8

5

8

15

15

C

D

C

D

15

15

(15)

(inf)

(15)

(30)

(a)

(b)

Figura 9.3: Terceiro e quarto passos no algoritmo de Dijkstra.

3. Actualiza-se a distância calculada dos nós adjacentes ao nó escolhido (os nós a que o nó escolhido está directamente ligado). Esta distância só é actualizada se a nova distância calculada for inferior à distância anteriormente calculada para o nó em causa.

Quando o algoritmo termina, todos os nós estão associados com a distância mínima ao nó original.

Na Figura 9.2 (b), mostramos o passo correspondente a seleccionar o primeiro nó, o nó A, actualizando-se a distância calculada para o nó B. Note-se que A não está ligado a F, mas F está ligado a A. Selecciona-se de seguida o nó B, actualizando-se as distâncias calculadas dos nós C, E e F (Figura 9.3 (a)).

Existem agora dois nós com a distância calculada de 15, selecciona-se arbitrariamente o nó C e actualiza-se a distância calculada associada ao nó D (Fi-

9.4. CAMINHOS MAIS CURTOS EM GRAFOS

249

(15)

(15)

F

F

15

25

25

5

20

15

5

20

(15)

(15)

A

10

B

15

E

(25)

A

10

B

15

E

(25)

(0)

(0)

5

8

5

8

15

15

C

D
C
D
15
15
(15)
(30)
(15)
(30)
(a)
(b)

Figura 9.4: Quinto e sexto passos no algoritmo de Dijkstra.

gura 9.3 (b)). Escolhe-se de seguida o nó F, o qual não leva à actualização de nenhuma distância calculada (Figura 9.4 (a)). O nó E é então escolhido, não levando também à actualização das distâncias calculadas (Figura 9.4 (b)). Finalmente, o nó D é escolhido, não originando a actualização de nenhuma distância calculada.

De modo a poder escrever este algoritmo em Python, teremos que escolher uma representação para grafos. Iremos escolher uma representação que é conhecida por lista de adjacências⁷, na qual um grafo é representado por um dicionário em que as chaves correspondem aos nós e cada nó está associado a uma lista de tuplos, contendo os nós ligados ao nó em questão e as respectivas distâncias.

No nosso exemplo, o nó B está ligado aos nós C, E e F, respectivamente, com as distâncias 5, 15 e 5, pelo que a sua representação será dada pelo par 'B':[('C', 5), ('E', 15), ('F', 5)]. A representação do grafo da Figura 9.1, que designaremos por g1, é:

```
g1 = { 'A':[( 'B', 10)], 'B':[( 'C', 5), ( 'E', 15), ( 'F', 5)],  
      'C':[( 'D', 15)], 'D':[( 'C', 15), ( 'E', 8)], 'E':[( 'F', 20)],  
      'F':[( 'A', 15), ( 'E', 25)]}
```


Assim, a função que implementa o algoritmo da distância mínima pode ser escrita em Python do seguinte modo:

```
def cdm (g, origem):
```

```
    nos = nos_do_grafo(g)
```

```
    Do inglês, “adjacency list”.
```

```
    250
```

CAPÍTULO 9. Dicionários

```
    dc = dist_inicial(nos, origem)
```

```
    while nos != []:
```

```
        n = dist_min(nos, dc) # selecciona o nó com menor dist^
```

```
        ancia
```

```
        remove(nos, n)
```

```
        for nn in g[n]: # para os nós a que nn está ligado actualiza_dist(n,
            nn[0], nn[1], dc)
```

```
    return dc
```

A função nos do grafo devolve a lista dos nós do grafo que é seu argumento.

Note-se a utilização de um ciclo for que, quando aplicado a um dicionário itera sobre as chaves do dicionário, tal como descrito na página 238.

```
def nos_do_grafo (g):
```

```
    res = []
```

```
    for el in g:
```

```
        res = res + [el]
```

```
    return res
```

A função dist inicial recebe como argumentos a lista dos nós do grafo e o nó inicial e devolve o dicionário que corresponde à distância inicial dos nós do grafo à origem, ou seja, zero para a origem e 'inf'

para os restantes nós. Para o grafo do nosso exemplo, esta função devolve o dicionário apresentado na página 247.

```
def dist_inicial (nos, origem):
```

```
    res = {}
```

```
    for n in nos:
```

```
        if n == origem:
```

```
            res[n] = 0
```

```
        else:
```

```
            res[n] = 'inf'
```

```
    return res
```

A função `dist_min` recebe a lista dos nós em consideração (`nos`) e o dicionário com as distâncias calculadas (`dc`) e devolve o nó com menor distância e a função `remove` remove o nó que é seu argumento da lista dos nós. Note-se que esta função utiliza a passagem por referência para alterar a lista `nos`.

```
def dist_min (nos, dc):
```

9.4. CAMINHOS MAIS CURTOS EM GRAFOS

251

```
n_min = nos[0]
```

```
# o primeiro nó
```

```
min = dc[n_min]
```

```
# a dist^
```

```
ancia associada ao primeiro nó
```

```
for n in nos:
```

```
    if menor(dc[n], min):
```

```
        min = dc[n]
```

```
# a dist^
```

ancia associada ao nó n

```
n_min = n
```

```
return n_min
```

```
def remove (nos, n):
```

```
for nn in range(len(nos)):
```

```
if nos[nn] == n:
```

```
del nos[nn]
```

```
return
```

A função actualiza_dist recebe um nó de partida (no partida), um nó a que o no partida está directamente ligado (no chegada), a distância entre o no partida e o no chegada e as distâncias calculadas (dc) e actualiza, nas distâncias calculadas, a distância associada ao no chegada.

```
def actualiza_dist (no_partida, no_chegada, dist, dc): dist_no_partida =  
dc[no_partida]
```

```
if menor(soma(dist_no_partida, dist), dc[no_chegada]): dc[no_chegada]  
= dist_no_partida + dist
```

Finalmente, as funções menor e soma, respectivamente comparam e somam dois valores, tendo em atenção que um deles pode ser 'inf'.

```
def menor (v1, v2):
```

```
if v1 == 'inf':
```

```
return False
```

```
elif v2 == 'inf':
```

```
return True
```

```
else:
```

```
return v1 < v2
```

252

```
def soma (v1, v2):
```

```
    if v1 == 'inf' or v2 == 'inf':
```

```
        return 'inf'
```

```
    else:
```

```
        return v1 + v2
```

Usando a função `cdm` e o grafo apresentado na Figura 9.1, podemos originar a seguinte interação:

```
> > > cdm(g1, 'A')
```

```
{'A': 0, 'C': 15, 'B': 10, 'E': 25, 'D': 30, 'F': 15}
```

```
> > > cdm(g1, 'B')
```

```
{'A': 20, 'C': 5, 'B': 0, 'E': 15, 'D': 20, 'F': 5}
```

9.5

Notas finais

Neste capítulo apresentamos o tipo dicionário, o qual corresponde ao conceito de lista associativa ou tabela de dispersão. Um dicionário associa chaves a valores, sendo os elementos de um dicionário indexados pela respectiva chave. Um dos aspectos importantes na utilização de dicionários corresponde a garantir que o acesso aos elementos de um dicionário é feito a tempo constante. Para estudar o modo como os dicionários são representados internamente no computador aconselhamos a consulta de [Cormen et al., 2009].

Apresentamos, como aplicação de utilização de dicionários, o algoritmo do caminho mais curto em grafos. O tipo grafo é muito importante em informática, estando para além da matéria deste livro. A especificação do tipo grafo e algumas das suas aplicações podem ser consultadas em [Dale and Walker, 1996].

Algoritmos que manipulam grafos podem ser consultados em [McConnell, 2008]

e em [Cormen et al., 2009].

9.6

Exercícios

1. Adicione ao processador de texto apresentado nos exercícios do Capítulo 8

a capacidade de produzir um índice alfabético. Para isso, algumas das pa-

9.6. EXERCÍCIOS

253

lavras do seu texto devem ser marcadas com um símbolo especial, por exemplo, o símbolo “@”, o qual indica que a palavra que o segue deve ser colocada no índice alfabético, que contém, para cada palavra, a página em que esta aparece. Por exemplo, se no seu texto aparece @programação, então a palavra “programação” deve aparecer no índice alfabético juntamente com a indicação da página em que aparece. As palavras do índice alfabético aparecem ordenadas alfabeticamente. Tenha em atenção os sinais de pontuação que podem estar juntos com as palavras mas não devem aparecer no índice.

Sugestão: utilize um dicionário em que cada chave contém uma palavra a inserir no índice alfabético, associado à página, ou páginas, em que esta aparece. Mantenha este dicionário ordenado.

2. Escreva um programa que aceite como dado de entrada um ficheiro fonte do seu formatador de texto, e conte quantas palavras e quantos parágrafos o texto contém. Tenha cuidado para não contar os comandos do formatador de texto como palavras.

Capítulo 10

Abstracção

ao de dados

‘Not very nice alone’ he interrupted, quite eagerly: ‘but you’ve no idea what a difference it makes, mixing it with other things’

Lewis Carroll, Through the Looking Glass

No Capítulo 3 introduzimos o conceito de abstracção procedimental e desde então temos utilizado a abstracção procedimental na definição de novas funções.

Já vimos que quase qualquer programa que efectue uma tarefa que não seja trivial, necessita de manipular entidades computacionais (dados) que correspondam a tipos estruturados de informação. Os programas que temos desenvolvido manipulam directamente instâncias dos tipos existentes em Python, sem criar novos tipos de informação. Mesmo em situações em que faria sentido a criação de novos tipos de informação, por exemplo, no cálculo do caminho mais curto em grafos apresentado na Secção 9.4, recorremos a uma representação de grafos e não definimos o tipo grafo, como seria natural.

Um programa complexo requer normalmente a utilização de tipos de informação que não existem na linguagem de programação utilizada. Analogamente ao que referimos no início do Capítulo 3, não é possível que uma linguagem de programação forneça todos os tipos de dados que são necessários para o desenvolvimento de uma certa aplicação. Torna-se assim importante fornecer ao programador a capacidade de definir novos tipos de dados dentro de um programa e de utilizar esses tipos de dados como se fossem tipos embutidos da linguagem de programação.

255

256

CAPÍTULO 10. ABSTRACÇÃO

~
,

AO DE DADOS

Neste capítulo, discutimos como criar tipos estruturados de informação que não estejam embutidos na nossa linguagem de programação, introduzindo o conceito de abstracção de dados.

10.1

A abstracção

ao em programação

ao

Sabemos que uma abstracção é uma descrição ou uma especificação simplificada de uma entidade que dá ênfase a certas propriedades dessa entidade e ignora outras. Uma boa abstracção especifica as

propriedades importantes e ignora os pormenores. Uma vez que as propriedades relevantes de uma entidade dependem da utilização a fazer com essa entidade, o termo “boa abstracção” está sempre associado a uma utilização particular da entidade.

A abstracção é um conceito essencial em programação. De facto, a actividade de programação pode ser considerada como a construção de abstracções que podem ser executadas por um computador.

Até aqui temos usado a abstracção procedimental para a criação de funções.

Usando a abstracção procedimental, os pormenores da realização de uma função podem ser ignorados, fazendo com que uma função possa ser substituída por uma outra função com o mesmo comportamento global, utilizando um outro algoritmo, sem que os programas que utilizam essa função sejam afectados.

A abstracção procedimental permite pois separar o modo como uma função é utilizada do modo como essa função é realizada.

O conceito equivalente à abstracção procedimental para dados (ou estruturas de informação) tem o nome de abstracção de dados. A abstracção de dados é uma metodologia que permite separar o modo como uma estrutura de informação é utilizada dos pormenores relacionados com o modo como essa estrutura de informação é construída a partir de outras estruturas de informação. Quando utilizamos um tipo de informação embutido em Python, por exemplo uma lista, não sabemos (nem queremos saber) qual o modo como o Python realiza internamente as listas. Para isso, recorreremos a um conjunto de operações embutidas (apresentadas na Tabela 5.1) para escrever as nossas funções. Se numa versão posterior do Python, a representação interna das listas for alterada, os nossos programas não são afectados por essa alteração. A abstracção de dados permite a obtenção de um comportamento semelhante para os dados criados no nosso

10.2. MOTIVAC

~
,

AO: NÚMEROS COMPLEXOS

abstracção procedimental, com a abstracção de dados, podemos substituir uma realização particular da entidade correspondente a um dado sem ter de alterar o programa que utiliza essa entidade, desde que a nova realização da entidade apresente o mesmo comportamento genérico, ou seja, desde que a nova realização corresponda, na realidade, à mesma entidade abstracta.

10.2

Motivação

ao: números complexos

A teoria dos tipos abstractos de informação considera que a definição de novos tipos de informação é feita em duas fases sequenciais, a primeira corresponde ao estudo das propriedades do tipo, e a segunda aborda os pormenores da realização do tipo numa linguagem de programação. A essência da abstracção de dados corresponde à separação das partes do programa que lidam com o modo como as entidades do tipo são utilizadas das partes que lidam com o modo como as entidades são representadas.

Para facilitar a nossa apresentação, vamos considerar um exemplo que servirá de suporte à nossa discussão. Suponhamos que desejávamos escrever um programa que lidava com números complexos. Os números complexos surgem em muitas formulações de problemas concretos e são entidades frequentemente usadas em Engenharia.

Os números complexos foram introduzidos, no século XVI, pelo matemático italiano Rafael Bombelli (1526–1572). Estes números surgiram da necessidade de calcular raízes quadradas de números negativos e são obtidos com a introdução do símbolo i , a unidade imaginária, que satisfaz a equação $i^2 = -1$. Um número complexo é um número da forma $a + bi$, em que tanto a , a parte real, como b , a parte imaginária, são números reais; um número complexo em que a parte real é nula chama-se um número imaginário puro. Sendo um número complexo constituído por dois números reais (à parte da unidade imaginária), pode-se estabelecer uma correspondência entre números complexos e pares de números reais, a qual está subjacente à representação de números complexos como pontos de um plano¹. Sabemos também que a soma, subtracção, multiplicação e

¹Chamado plano de Argand, em honra ao matemático francês Jean-Robert Argand (1768–

1822) que o introduziu em 1806.

258

CAPÍTULO 10. ABSTRACC

~
,

AO DE DADOS

divis~ao de números complexos s~ao definidas do seguinte modo: $(a + bi) + (c + di) = (a + c) + (b + d)i$

$$(a + bi) - (c + di) = (a - c) + (b - d)i$$

$$(a + bi).(c + di) = (ac - bd) + (ad + bc)i \quad (a + bi) = ac + bd + bc - ad$$

(

i

$$c + di)$$

$$c^2 + d^2$$

$$c^2 + d^2$$

operaç~oes que s~ao óbvias, tendo em atenç~ao que $i^2 = -1$.

Suponhamos que desejávamos escrever um programa que iria lidar com números complexos. Este programa deverá ser capaz de adicionar, subtrair, multiplicar e dividir números complexos, deve saber comparar números complexos, *etc.*

A hipótese de trabalharmos com números complexos com a parte real e a parte imaginária como entidades separadas é impensável pela complexidade que irá introduzir no nosso programa e no nosso raciocínio, pelo que esta hipótese será imediatamente posta de lado.

A primeira tentativa de um programador inexperiente será a de começar por pensar em como representar números complexos com base nas estruturas de informação que conhece. Por exemplo, esse programador poderá decidir que um número complexo é representado por um tuplo, em que o primeiro elemento contém a parte real e o segundo elemento contém a parte imaginária. Assim o complexo $a + bi$ será representado pelo tuplo (a, b) . Esse programador, tendo duas

variáveis que correspondem a números complexos, `c1` e `c2`, irá naturalmente escrever a seguinte expressão para somar esses números, dando origem ao número complexo `c3`:

```
c3 = (c1[0] + c2[0], c1[1] + c2[1]).
```

Embora a expressão anterior esteja correcta, ela introduz uma complexidade desnecessária no nosso raciocínio. Em qualquer operação que efectuemos sobre números complexos temos que pensar simultaneamente na operação matemática e na representação que foi escolhida para complexos.

Suponhamos, como abrogagem alternativa, e utilizando uma estratégia de pensamento positivo, que existem em Python as seguintes funções:

10.2. MOTIVAC

~
,

AO: NÚMEROS COMPLEXOS

259

`cria_compl(r, i)` esta função recebe como argumentos dois números reais, `r` e `i`, e tem como valor o número complexo cuja parte real é `r` e cuja parte imaginária é `i`.

`p_real(c)` esta função recebe como argumento um número complexo, `c`, e tem como valor a parte real desse número.

`p_imag(c)` esta função recebe como argumento um número complexo, `c`, e tem como valor a parte imaginária desse número.

Com base nesta suposição, podemos escrever funções que efectuam operações aritméticas sobre números complexos, embora não saibamos como estes são representados. Assim, podemos escrever as seguintes funções que, respectivamente, adicionam, subtraem, multiplicam e dividem números complexos: `def soma_compl (c1, c2):`

```
p_r = p_real(c1) + p_real(c2)
```

```
p_i = p_imag(c1) + p_imag(c2)
```

```
return cria_compl(p_r, p_i)
```

```
def subtrai_compl (c1, c2):
```

```
p_r = p_real(c1) - p_real(c2)
```

```
p_i = p_imag(c1) - p_imag(c2)
```

```
return cria_compl(p_r, p_i)
```

```
def multiplica_compl (c1, c2):
```

```
p_r = p_real(c1) p_real(c2) - p_imag(c1) p_imag(c2) p_i = p_real(c1)
p_imag(c2) + p_imag(c1) p_real(c2) return cria_compl(p_r, p_i)
```

```
def divide_compl (c1, c2):
```

```
den = p_real(c2) p_real(c2) + p_imag(c2) p_imag(c2) p_r = (p_real(c1)
p_real(c2) + p_imag(c1) p_imag(c2))/den p_i = (p_imag(c1) p_real(c2)
- p_real(c1) p_imag(c2))/den return cria_compl(p_r, p_i)
```

Independentemente da forma como será feita a representação de números complexos, temos também interesse em dotar o nosso programa com a possibilidade de mostrar um número complexo tal como habitualmente o escrevemos:

260

CAPÍTULO 10. ABSTRACC

~
,

AO DE DADOS

o número complexo com parte real a e parte imaginária b é apresentado ao mundo exterior² como $a + b i$. Note-se que esta representação de complexos é exclusivamente “para os nossos olhos”, no sentido de que ela nos permite visualizar números complexos mas não interfere no modo como o programa lida com números complexos. A este tipo de representação chama-se representação externa, em oposição à representação interna, a qual é utilizada pelo Python para lidar com números complexos (e que é desconhecida neste momento).

Podemos escrever a seguinte função para produzir uma representação externa para números complexos:

```
def escreve_compl (c):
```

```
if p_imag(c) >= 0:
```

```
print(p_real(c),'+', p_imag(c),'i')
```

else:

```
print(p_real(c),'-', abs(p_imag(c)), 'i')
```

De modo a que as funções anteriores possam ser utilizadas em Python é necessário que concretizemos as operações cria compl, p real e p imag. Para isso, teremos de encontrar um modo de agregar a parte real com a parte imaginária de um complexo numa única entidade.

Suponhamos que decidimos representar um número complexo por um tuplo, em que o primeiro elemento contém a parte real e o segundo elemento contém a parte imaginária. Com base nesta representação, podemos escrever as funções cria compl, p real e p imag do seguinte modo: def cria_compl (r, i):

```
return (r, i)
```

```
def p_real (c):
```

```
return c[0]
```

```
def p_imag (c):
```

```
return c[1]
```

A partir deste momento podemos utilizar números complexos, como o demonstra a seguinte interacção:

2Por “mundo exterior” entenda-se o mundo com que o Python comunica.

10.2. MOTIVAC

, ~

AO: NÚMEROS COMPLEXOS

261

```
>>> c1 = cria_compl(3, 5)
```

```
>>> p_real(c1)
```

3

```
> > > p_imag(c1)
```

```
5
```

```
> > > escreve_compl(c1)
```

```
3 + 5 i
```

```
> > > c2 = cria_compl(1, -3)
```

```
> > > escreve_compl(c2)
```

1 - 3 i

```
> > > c3 = soma_compl(c1, c2)
```

```
> > > escreve_compl(c3)
```

4 + 2 i

```
> > > escreve_compl(subtrai_compl(cria_compl(2, -8), c3))
```

-2 - 10 i

Podemos agora imaginar que alguém objecta à nossa representação de números complexos como tuplos, argumentando que esta não é a mais adequada pois os índices 0 e 1, usados para aceder aos constituintes de um complexo, nada dizem sobre o constituinte acedido. Segundo essa argumentação, seria mais natural utilizar um dicionário para representar complexos, usando-se o índice

'r' para a parte real e o índice 'i' para a parte imaginária. Motivados por esta argumentação, podemos alterar a nossa representação, de modo a que um complexo seja representado por um dicionário.

Podemos dizer que o número complexo $a + bi$ é representado pelo dicionário com uma chave 'r', cujo valor é a e e uma chave 'i', cujo valor é b. Utilizando a notação $\diamond[X]$ para indicar a representação interna da entidade X, podemos escrever, $\diamond[a + bi] = \{'r':a,$

'i':b\}.

Com base na representação utilizando dicionários, podemos escrever as seguintes funções para realizar as operações que criam números complexos e seleccionam os seus componentes:

```
def cria_compl (r, i):
```

```
    return {'r':r, 'i':i}
```

```
def p_real (c):
```

```
    return c['r']
```

262

CAPÍTULO 10. ABSTRACC

, ~

AO DE DADOS

```
def p_imag (c):
```

```
    return c['i']
```

Utilizando esta representação, e recorrendo às operações definidas nas páginas 259 e seguintes, podemos replicar a interacção obtida

com a representação de complexos através de tuplos (apresentada na página 260) sem qualquer alteração. Este comportamento revela a independência entre as funções que efectuam operações aritméticas sobre números complexos e a representação interna de números complexos. Este comportamento foi obtido através de uma separação clara entre as operações que manipulam números complexos e as operações que correspondem à definição de complexos (cria compl, p real e p imag). Esta separação permite-nos alterar a representação de complexos sem ter de alterar o programa que lida com números complexos.

Esta é a essência da abstracção de dados, a separação entre as propriedades dos dados e os pormenores da realização dos dados numa linguagem de programação.

Esta essência é traduzida pela separação das partes do programa que lidam com o modo como os dados são utilizados das partes que lidam com o modo como os dados são representados.

10.3

Tipos abstractos de informação

ao

Nesta secção apresentamos os passos a seguir no processo de criação de novos tipos de informação. Sabemos que um tipo de informação é uma colecção de entidades chamadas elementos do tipo, conjuntamente com uma colecção de operações que podem ser efectuadas sobre essas entidades. Estas operações constroem novas entidades, seleccionam constituintes dessas entidades, identificam e comparam elementos dessas entidades.

A metodologia que apresentamos tem o nome de metodologia dos tipos abstractos de informação³, e a sua essência é a separação das partes do programa que lidam com o modo como as entidades do tipo são utilizadas das partes que lidam com o modo como as entidades são representadas.

Na utilização da metodologia dos tipos abstractos de informação devem ser 3Em inglês “abstract data types” ou ADTs.

10.3. TIPOS ABSTRACTOS DE INFORMAC

seguidos quatro passos sequenciais: (1) a identificação das operações básicas; (2) a axiomatização das operações básicas; (3) a escolha de uma representação para os elementos do tipo; e (4) a concretização das operações básicas para a representação escolhida.

Esta metodologia permite a definição de tipos de informação que são independentes da sua representação. Esta independência leva à designação destes tipos por tipos abstractos de informação.

Nas próximas secções exemplificamos estes passos para a definição do tipo complexo.

10.3.1

Identificação

ao das operações

oes básicas

Para definir a parte do programa que lida com o modo como os dados são utilizados, devemos identificar, para cada tipo de dados, o conjunto das operações básicas que podem ser efectuadas sobre os elementos desse tipo. Recorde-se que um tipo de informação é caracterizado por um conjunto de entidades, os elementos do tipo, e por um conjunto de operações aplicáveis a essas entidades.

É evidente que ao definir um tipo de informação não podemos definir todas as operações que manipulam os elementos do tipo. A ideia subjacente à definição de um novo tipo é a definição do mínimo possível de operações que permitam caracterizar o tipo. Estas operações são chamadas as operações básicas e dividem-se em quatro grupos, os construtores, os selectores, os reconhecedores e os testes.

1. Os construtores são operações que permitem construir novos elementos do tipo.

2. Os selectores são operações que permitem aceder (isto é, seleccionar) aos constituintes dos elementos do tipo.

3. Os reconhecedores são operações que identificam elementos do

tipo. Os reconhecedores são de duas categorias. Por um lado, fazem a distinção entre os elementos do tipo e os elementos de qualquer outro tipo, reconhecendo explicitamente os elementos que pertencem ao tipo. Por outro lado, identificam elementos do tipo que se individualizam dos restantes por possuírem certas propriedades particulares.

264

CAPÍTULO 10. ABSTRACC

~
s

AO DE DADOS

4. Os testes são operações que efectuam comparações entre os elementos do tipo.

O papel das operações básicas é construir elementos do tipo (os construtores), seleccionar componentes dos elementos do tipo (os selectores) e responderem a perguntas sobre os elementos do tipo (os reconhecedores e os testes).

Ao conjunto das operações básicas para um dado tipo dá-se o nome de assinatura do tipo. De modo a enfatizar que as operações básicas são independentes da linguagem de programação utilizada (na realidade, estas especificam de um modo abstracto o que é o tipo), é utilizada a notação matemática para as caracterizar.

Para o tipo complexo podemos considerar as seguintes operações básicas: 1. Construtores:

• $\text{cria compl} : \text{real} \times \text{real} \rightarrow \text{complexo}$

$\text{cria compl}(r, i)$ tem como valor o número complexo cuja parte real é r e cuja parte imaginária é i .

2. Selectores:

- $\text{p real} : \text{complexo} \rightarrow \text{real}$

$\text{p real}(c)$ tem como valor a parte real do complexo c .

- $\text{p imag} : \text{complexo} \rightarrow \text{real}$

$\text{p imag}(c)$ tem como valor a parte imaginária do complexo c .

3. Reconhecedores:

- complexo : universal $\diamond \rightarrow$ lógico

complexo(arg) tem o valor verdadeiro se arg é um número complexo e tem o valor falso em caso contrário.

- compl zero : complexo $\diamond \rightarrow$ lógico

compl zero(c) tem o valor verdadeiro se c é o complexo $0 + 0i$ e tem o valor falso em caso contrário.

- imag puro : complexo $\diamond \rightarrow$ lógico

imag puro(c) tem o valor verdadeiro se c é um imaginário puro, ou seja, um complexo da forma $0 + bi$, e tem o valor falso em caso contrário.

10.3. TIPOS ABSTRACTOS DE INFORMAC

~
,

AO

4. Testes:

• compl iguais : complexo $\times$ complexo $\rightarrow$ lógico compl iguais(c1, c2) tem o valor verdadeiro se c1 e c2 correspondem ao mesmo número complexo e tem o valor falso em caso contrário.

Assim, a assinatura do tipo complexo é:

cria compl : real $\times$ real $\rightarrow$ complexo

p real : complexo $\rightarrow$ real

p imag : complexo $\rightarrow$ real

complexo : universal $\rightarrow$ lógico

compl zero : complexo $\rightarrow$ lógico

imag puro : complexo $\rightarrow$ lógico

compl iguais : complexo $\times$ complexo $\rightarrow$ lógico Deveremos ainda definir uma representação externa para complexos, por exemplo, podemos definir que a notação correspondente aos elementos do tipo complexo é da forma $a + b i$, em que a e b são reais. Com base nesta representação, devemos especificar os transformadores de entrada e de saída.

O transformador de entrada transforma a representação externa para as entidades abstractas na sua representação interna (seja ela qual for) e o transformador de saída transforma a representação interna das entidades na sua representação externa. Neste capítulo, não especificaremos o transformador de entrada para os tipos que definimos. O transformador de saída para números complexos, ao qual chamamos escreve compl, recebe uma entidade do tipo complexo e escreve essa entidade sob a forma $a + b i$.

Ao especificarmos as operações básicas e os transformadores de entrada e de saída para um dado tipo, estamos a criar uma extensão conceptual da nossa linguagem de programação como se o tipo fosse embutido. Podemos então escrever programas que manipulam entidades do novo tipo, em termos dos construtores, selectores, reconhecedores, testes e transformadores de entrada e de saída, mesmo antes de termos escolhido uma representação para o tipo e de termos es-

CAPÍTULO 10. ABSTRACC

AO DE DADOS

critérios que correspondam às operações básicas. Deste modo, obtemos uma verdadeira separação entre a utilização do tipo de informação e a sua realização através de um programa.

10.3.2

Axiomatização

ao

A axiomatização especifica o modo como as operações básicas se relacionam entre si.

Para o caso dos números complexos, sendo r e i números reais, esta axiomatização é dada por:

$\text{complexo}(\text{cria compl}(r, i)) = \text{verdadeiro}$

$\text{compl zero}(c) = \text{compl iguais}(c, \text{cria compl}(0, 0))$
 $\text{imag puro}(\text{cria compl}(0, b)) = \text{verdadeiro}$

$\text{p real}(\text{cria compl}(r, i)) = r$

$\text{p imag}(\text{cria compl}(r, i)) = i$

♦ c se $\text{complexo}(c)$

$\text{cria compl}(\text{p real}(c), \text{p imag}(c)) =$

$\perp$ em caso contrário

$\text{compl iguais}(\text{cria compl}(x, y), \text{cria compl}(w, z)) = (x = w) \wedge (y = z)$

Neste passo especificam-se as relações obrigatoriamente existentes entre as operações básicas, para que estas definam o tipo de um modo coerente.

10.3.3

Escolha da representação

O terceiro passo na definição de um tipo de informação consiste em escolher uma representação para os elementos do tipo em termos de outros tipos existentes.

No caso dos números complexos, iremos considerar que o complexo $a + bi$ é representado por um dicionário em que o valor da chave 'r' é a parte real e o valor da chave 'i' é a parte imaginária, assim $\Diamond[a + bi] = \{r':\Diamond[a], i':\Diamond[b]\}$.

O símbolo $\perp$ representa indefinido.

10.3. TIPOS ABSTRACTOS DE INFORMAÇÃO

~
,

AO

267

10.3.4

Realização

ao das operações

ões básicas

O último passo na definição de um tipo de informação consiste em realizar as operações básicas definidas no primeiro passo em termos da representação definida no terceiro passo. É evidente que a realização destas operações básicas deve verificar a axiomatização definida no segundo passo.

Para os números complexos, teremos:

1. Construtores:

```
def cria_complexo (r, i):
```

```
    if numero(r) and numero(i):
```

```
        return {'r':r, 'i':i}
```

```
    else:
```

```
        raise ValueError ('cria_complexo: argumento errado') A função de  
tipo lógico, numero tem o valor True apenas se o seu argumento for  
um número (int ou float). Esta função é definida do seguinte modo:
```

```
def numero(x):
```

```
    return isinstance(x, (int, float))
```

Recorde-se que a função isinstance determina se o tipo do seu primeiro argumento pertence ao tuplo que é seu segundo argumento. No caso da função numero, queremos garantir que o seu argumento é ou um número inteiro (int) ou um número real (float).

2. Selectores:

```
def p_real (c):
```

```
    return c['r']
```

```
def p_imag (c):
```

```
    return c['i']
```

3. Reconhecedores:

268

CAPÍTULO 10. ABSTRACC

~
,

AO DE DADOS

def complexo (c):

if isinstance(c, (dict)):

if len(c) == 2 and 'r' in c and 'i' in c:

if numero(c['r']) and numero(c['i']):

return True

else:

return False

else:

return False

else:

return False

Convém fazer uma observação em relação à função complexo. Esta função, sendo um reconhecedor, pode receber qualquer tipo de argumento, de-vendo ter o valor True apenas se o seu argumento é um número complexo.

Assim, o primeiro teste desta função verifica se o seu argumento é um dicionário, `isinstance(c, (dict))`. Se o for, sabemos que podemos calcular o seu comprimento, `len(c)`, e determinar se as chaves 'r' e 'i'

existem no dicionário, `'r' in c and 'i' in c`. Só após todos estes testes deveremos verificar se os valores associados às chaves são números.

def compl_zero (c) :

```
return zero(c['r']) and zero(c['i'])
```

```
def imag_puro (c):
```

```
return zero(c['r'])
```

As funções `compl zero` e `imag puro` utilizam a função `zero` que recebe como argumento um número e tem o valor `True` se esse número for zero, tendo o valor `False` em caso contrário. A definição da função `zero` é necessária, uma vez que os componentes de um número complexo são números reais, e o teste para determinar se um número real é zero deve ser traduzido por um teste de proximidade em valor absoluto do valor zero. Por esta razão, a função `zero` poderá ser definida como: `def zero (x):`

```
return abs(x) < 0.0001
```

5 Testes semelhantes já foram utilizados nas Secções 3.4.5, 6.2.1 e 6.2.2.

10.4. BARREIRAS DE ABSTRACC

~
s

AO

269

em que 0.0001 corresponde ao valor do erro admissível.

4. Testes:

```
def compl_iguais (c1, c2) :  
    return igual(c1['r'], c2['r']) and \  
    igual(c1['i'], c2['i'])
```

A função `compl_iguais`, utiliza o predicado `iguais` para testar a igualdade de dois números reais. Se 0.0001 corresponder ao erro admissível, esta função é definida por:

```
def igual (x, y):  
    return abs(x - y) < 0.0001
```

No passo correspondente à realização das operações básicas, devemos também especificar os transformadores de entrada e de saída. Neste capítulo, apenas nos preocupamos com o transformador de saída, o qual é traduzido pela função `escreve` apresentada na página 260.

10.4

Barreiras de abstracção

ao

Depois de concluídos todos os passos na definição de um tipo abstracto de informação (a definição de como os elementos do tipo são utilizados e a definição de como eles são representados, bem como a escrita de funções correspondentes às respectivas operações), podemos juntar o conjunto de funções correspondente ao tipo a um programa que utiliza o tipo como se este fosse um tipo embutido na linguagem. O programa acede a um conjunto de operações que são específicas do tipo e que, na realidade, caracterizam o seu comportamento como tipo de informação. Qualquer manipulação efectuada sobre uma entidade de um dado tipo deve apenas recorrer às operações básicas para esse tipo.

Ao construir um novo tipo abstracto de informação estamos a criar uma nova camada conceptual na nossa linguagem, a qual corresponde ao tipo definido.

Esta camada é separada da camada em que o tipo não existe por

barreiras de abstracção. Conceptualmente, estas barreiras impedem qualquer acesso aos elementos do tipo que não seja feito através das operações básicas. Na Figura 10.1

270

CAPÍTULO 10. ABSTRACC

AO DE DADOS

números complexos com

partes real e imaginária

cria compl p real p imag

complexo compl zero imag puro

compl iguais escreve complexo

números complexos como dicionários

Operaç

ao de acesso com chave

operacionalizaç

ao interna de

dicionários

(seja ela qual for)

Figura 10.1: Camadas de abstracção na definição de complexos.

apresentamos as camadas e as barreiras de abstracção existentes num programa depois da definição do tipo complexo.

Em cada uma destas camadas, a barreira de abstracção separa os programas que usam a abstracção de dados (que estão situados acima da barreira) dos programas que realizam a abstracção de dados (que estão situados abaixo da barreira).

Para ajudar a compreender a necessidade da separação destas camadas, volte-mos a considerar o exemplo dos números complexos.

Suponhamos que, com base na representação utilizando dicionários, escrevamos as operações básicas para os números complexos. Podemos agora utilizar o módulo correspondente ao tipo complexo num programa que manipule números complexos. De acordo com a metodologia estabelecida para os tipos abstractos de informação, apenas devemos aceder aos elementos do tipo complexo através das suas operações básicas (as operações disponíveis para uso público). Ou seja, embora possamos saber que os complexos são representados através de dicionários, não somos autorizados a manipulá-los através da referência à chave associada a cada elemento de um complexo.

Definindo os números complexos através de funções, como o fizemos na secção anterior, nada na linguagem nos impede de manipular directamente a sua representação. Ou seja, se um programador tiver conhecimento do modo como um

10.4. BARREIRAS DE ABSTRACC

~
,

AO

271

tipo é representado, nada o impede de violar a regra estabelecida na metodologia e aceder directamente aos componentes do tipo através da manipulação da estrutura por que este é representado. Por exemplo, se $c1$ e $c2$ corresponderem a números complexos, esse programador poderá pensar em adicionar directamente $c1$ e $c2$, através de:

$\{r:(c1[r] + c2[r]), i:(c1[i] + c2[i])\}$.

No entanto, esta decisão corresponde a uma má prática de programação, a qual deve ser sempre evitada, pelas seguintes razões:

1. A manipulação directa da representação do tipo faz com que o programa seja dependente dessa representação. Suponhamos que, após o desenvolvimento do programa, decidamos alterar a representação de números complexos de dicionários para outra representação qualquer. No caso da regra da metodologia ter sido violada, ou seja se uma barreira de abstracção tiver sido “quebrada”, o programador teria que percorrer todo o programa e alterar todas as manipulações directas da estrutura que representa o tipo.

2. O programa torna-se mais difícil de escrever e de compreender, uma vez que a manipulação directa da estrutura subjacente faz com

que se perca o nível de abstracção correspondente à utilização do tipo.

As linguagens de programação desenvolvidas antes do aparecimento da metodologia para os tipos abstractos de informação não possuem mecanismos para garantir que toda a utilização dos elementos de um dado tipo é efectuada recorrendo exclusivamente às operações específicas desse tipo. As linguagens de programação mais recentes (por exemplo, Ada, CLU, Mesa, Modula-2 e Python) fornecem mecanismos que garantem que as manipulações efectuadas sobre os elementos de um tipo apenas utilizam as operações básicas desse tipo. Este comportamento é obtido através da utilização de dois conceitos chamados encapsulação da informação e anonimato da representação.

A encapsulação da informação corresponde ao conceito de que o conjunto de funções que corresponde ao tipo de informação engloba toda a informação referente ao tipo. Estas funções estão representadas dentro de um módulo que está protegido dos acessos exteriores. Este módulo comporta-se de certo modo como o inglês, “data encapsulation”.

272

CAPÍTULO 10. ABSTRACC

~
,

AO DE DADOS

interface

representação

Figura 10.2: Módulo correspondente a um tipo abstracto de informação.

um bloco, com a diferença que “exporta” certas operações, permitindo apenas o acesso às operações exportadas. O anonimato da representação corresponde ao conceito de que este módulo guarda como segredo o modo escolhido para representar os elementos do tipo. O único acesso que o programa tem aos elementos do tipo é através das operações básicas definidas dentro do módulo que corresponde ao tipo. Na Figura 10.2 apresentamos de uma forma esquemática o módulo conceptual correspondente a um tipo abstracto de informação. Este módulo contém uma parte que engloba a

representação do tipo, representação essa que está escondida do resto do programa, e um conjunto de funções que efectuam a comunicação (interface) entre o programa e a representação interna.

A vantagem das linguagens que incorporam os mecanismos da metodologia dos tipos abstractos de informação reside no facto de estas proibirem efectivamente a utilização de um tipo abstracto através da manipulação directa da sua representação: toda a informação relativa ao tipo está contida no módulo que o define, o qual guarda como segredo a representação escolhida para o tipo.

10.5

Objectos

Os módulos que implementam tipos abstractos de informação correspondem em Python ao conceito de objecto. Um objecto é uma entidade computacional que apresenta, não só informação interna (a representação de um elemento de um tipo), como também um conjunto de operações que definem o seu comporta-7Do inglês, “information hiding”.

10.5. OBJECTOS

273

mento. O conceito de objecto foi introduzido com a linguagem SIMULA8, foi vulgarizado em 1984 pelo sistema operativo do Macintosh, foi adoptado pelo sistema operativo Windows, uma década mais tarde, e está hoje na base de várias linguagens de programação, por exemplo, o C + +, o CLOS, o Java e o Python.

A manipulação de objectos não é baseada em funções que calculam valores, mas sim no conceito de uma entidade que recebe solicitações e reage apropriadamente a essas solicitações, recorrendo à informação interna do objecto. As funções associadas a um objecto são vulgarmente designadas por métodos. É importante distinguir entre solicitações e métodos: uma solicitação é um pedido feito a um objecto; um método é a função utilizada pelo objecto para processar essa solicitação.

Para definir objectos em Python, criamos um módulo com o nome do objecto, associando-lhe os métodos correspondentes e esse objecto. Em notação BNF, um objecto em Python é definido do seguinte modo9:

◆definição de objecto◆ ::= class ◆nome◆ {(◆nome◆)}: CR

TAB ♦definição de método♦ +

♦definição de método♦ ::= def ♦nome♦ (self{, ♦parâmetros formais♦}): CR

TAB ♦corpo♦

Os símbolos não terminais ♦nome♦, ♦parâmetros formais♦ e ♦corpo♦ foram definidos nos capítulos 2 e 3.

Nesta definição:

1. a palavra class (um símbolo terminal) indica que se trata da definição de um objecto. Existem duas alternativas para esta primeira linha: (1) pode apenas ser definido um nome, ou (2) pode ser fornecido um nome seguido de outro nome entre parênteses. Neste capítulo apenas consideramos a primeira alternativa, sendo a segunda considerada no Capítulo 11. O

nome imediatamente após a palavra class representa o nome do objecto que está a ser definido;

2. A ♦definição de método♦ corresponde à definição de um método que está associado ao objecto. Podem ser definidos tantos métodos quantos se 8[Dahl and Nygaard, 1967].

9Esta definição corresponde em Python a uma ♦definição♦.

274

CAPÍTULO 10. ABSTRACC

~
,

AO DE DADOS

desejar, mas existe sempre pelo menos um método cujo nome é init (este aspecto não está representado na nossa expressão em notação BNF); 3. Os parâmetros formais de um método contêm sempre, como primeiro elemento, um parâmetro com o nome self¹⁰.

Os objectos contêm informação interna. Esta informação interna é caracterizada por uma ou mais variáveis. Os nomes destas variáveis utilizam a notação de

♦nome composto♦ apresentada na página 105, em que o primeiro nome é sempre self. Por exemplo, se dentro de um método

desejássemos utilizar uma variável que normalmente designaríamos com o nome `r`, esta variável deveria ser escrita através do nome composto `self.r`.

De acordo com a definição anterior, consideremos a seguinte definição de um objecto com o nome `compl11` correspondente a um número complexo: `class compl`:

```
def __init__(self, real, imag):  
    if isinstance(real, (int, float)) and \   
        isinstance(imag, (int, float)):  
        self.r = real  
        self.i = imag  
    else:  
        raise ValueError('complexo: argumento errado')  
def p_real(self):  
    return self.r  
def p_imag(self):  
    return self.i  
def compl_zero(self):  
    return self.r == 0 and self.i == 0
```

10A palavra inglesa “self” corresponde à individualidade ou identidade de uma pessoa ou de uma coisa.

11Por simplicidade de apresentação, utilizamos o operador relacional `==` para comparar os elementos de um número complexo, quando, na realidade, deveríamos utilizar os predicados zero (apresentado na página 268) e igual (apresentado na página 269).

10.5. OBJECTOS

275

```
def imag_puro(self):  
    return self.r == 0
```

```

def compl_iguais (self, outro):

return self.r == outro.p_real() and \

self.i == outro.p_imag()

def escreve (self):

if self.i >= 0:

print(self.r,'+', self.i,'i')

else:

print(self.r,'-', abs(self.i),'i')

```

A execução desta definição pelo Python dá origem à criação do objecto `compl`, estando este objecto associado a métodos que correspondem aos nomes `init`, `p_real`, `p_imag`, `compl_zero`, `imag_puro`, `compl_iguais` e `escreve`.

A partir do momento em que um objecto é criado, passa a existir uma nova função em Python cujo nome corresponde ao nome do objecto e cujos parâmetros correspondem a todos os parâmetros formais da função `init` que lhe está associada, com excepção do parâmetro `self`. O comportamento desta função é definido pela função `init` associada ao objecto. No nosso exemplo, esta é a função `compl` de dois argumentos. A chamada à função `compl(3, 2)` devolve o objecto correspondente ao número complexo $3 + 2i$. Podemos, assim, originar a seguinte interacção:

```

>>> compl

<class '__main__.compl'>

>>> c1 = compl(3, 2)

>>> c1

<__main__.compl object at 0x11fab90>

```

As duas primeiras linhas desta interacção mostram que `compl` é uma classe (um novo tipo de entidades computacionais que existem nos nossos programas) e as duas últimas linhas mostram que o nome `c1` está associado a um objecto, localizado na posição de memória `0x11fab90`.

O resultado da execu~ao da instru~ao `c1 = compl(3, 2)`, foi a cria~ao do ob-

276

CAPÍTULO 10. ABSTRACC

~
,

AO DE DADOS

`p_real`

`p_imag`

`c1`

`compl_zero imag_puro`

`self.r`

3

`self.i`

2

`compl_iguais` escreve

Figura 10.3: Objecto correspondente ao número complexo $3 + 2i$.

jecto apresentado na Figura 10.3 e a associa~ao do nome `c1` a este objecto. Este objecto “esconde” do exterior a representa~ao do seu número complexo (dado pelos valores das variáveis `self.r` e `self.i`), fornecendo seis métodos para inter-actuar com o número complexo, cujos nomes s~ao `p_real`, `p_imag`, `compl_zero`, `imag_puro`, `compl_iguais` e `escreve`.

Os métodos associados a um objecto correspondem a func~oes que manipulam as variáveis associadas ao objecto, variáveis essas que est~ao guardadas dentro do objecto, correspondendo a um ambiente local ao objecto, o qual existe enquanto o objecto correspondente existir. Para invocar os métodos associados a um objecto utiliza-se a notaç~ao de $\blacklozenge$ nome composto $\blacklozenge$ apresentada na página 105, em que o nome é composto pelo nome do objecto, seguido do nome do método¹².

Por exemplo `c1.p_real` é o nome do método que devolve a parte real do complexo correspondente a `c1`. Para invocar a execução de um método, utilizam-se todos os parâmetros formais da função correspondente ao método com excepção do parâmetro `self`. Assim, podemos originar a seguinte interacção:

```
> > > c1 = compl(3, 2)
```

```
> > > c1.p_real()
```

```
3
```

```
> > > c1.p_imag()
```

```
2
```

```
> > > c1.escreve()
```

```
3 + 2 i
```

```
> > > c2 = compl(2, -2)
```

```
> > > c2.escreve()
```

2 - 2 i

12Note-se que as funções que manipulam ficheiros apresentadas no Capítulo 8 e a função lower apresentada na página 240 do Capítulo 9, na realidade correspondem a métodos.

10.5. OBJECTOS

277

Na interacção anterior criámos dois objectos `c1` e `c2` correspondentes a números complexos. Cada um destes objectos armazena a parte inteira e a parte imaginária de um número complexo, apresentando comportamentos semelhantes, apenas diferenciados pela sua identidade, um corresponde ao complexo $3 + 2i$ e o outro ao complexo $2 - 2i$.

É importante fazer duas observações em relação à nossa definição do objecto `compl`. Em primeiro lugar, não definimos um reconhecedor correspondente à operação básica complexo. Na realidade, ao criar um objecto, o Python associa automaticamente o nome do objecto à entidade que o representa. A função embutida `type` devolve o tipo do seu argumento, como o mostra a seguinte interacção:

```
>>> c1 = compl(3, 5)
```

```
>>> type(c1)
```

```
<class '__main__.compl'>
```

A função `isinstance`, introduzida na página 267, permite determinar se uma dada entidade corresponde a um complexo:

```
>>> c1 = compl(9, 6)
```

```
>>> isinstance(c1, compl)
```

True

Existe assim, uma solicitação especial que pode ser aplicada a qualquer objecto e que pede a identificação do objecto. Por esta razão, ao criarmos um tipo abstracto de informação recorrendo a objectos, não necessitamos de escrever o reconhecedor que distingue os elementos do tipo dos restantes elementos.

A segunda observação respeita o método `compl iguais`. Na nossa definição das operações básicas para complexos, definimos a função `compl iguais` que permite decidir se dois complexos são iguais. Na nossa implementação, o método `compl iguais` apenas tem um argumento correspondente a um complexo. Este método compara o complexo correspondente ao seu objecto com outro complexo qualquer. Assim, por omissão, o primeiro argumento da comparação com outro complexo é o complexo correspondente ao objecto em questão, como a seguinte interacção ilustra:

```
> > > c1 = compl(9, 6)
```

278

CAPÍTULO 10. ABSTRACC

~
,

AO DE DADOS

```
> > > c2 = compl(7, 6)
```

```
> > > c1.compl_iguais(c2)
```

False

```
> > > c2.compl_iguais(c1)
```

False

Em programação com objectos existe uma abstracção chamada classe. Uma classe corresponde a uma infinidade de objectos com as mesmas variáveis e com o mesmo comportamento. É por esta razão, que na definição de um objecto o Python se usa a palavra `class` (ver a definição apresentada na página 273). Por exemplo, os complexos correspondem a uma classe de objectos que armazenam uma parte real, `self.r`, e uma parte imaginária, `self.i`, e cujo comportamento é definido pelas funções

`init`, `p real`, `p imag`, `compl zero`, `imag puro`, `compl iguais` e `escreve`. Os complexos `c1` e `c2` criados na interacção anterior correspondem a instâncias do objecto `compl`, sendo conhecidos como instâncias da classe `compl`.

Uma classe é um potencial gerador de instâncias, objectos cujas variáveis e comportamento são definidos pela classe. São as

instâncias que contêm os valores das variáveis associadas à classe e que reagem a solicitações – a classe apenas define como são caracterizadas as variáveis e o comportamento das suas instâncias.

As classes podem ser representadas graficamente como se indica na Figura 10.4.

Uma classe corresponde a um rectângulo com três partes, contendo, respectivamente, o nome da classe, o nome das variáveis e o nome dos métodos. As instâncias são caracterizadas pelo seu nome (dentro de um rectângulo) e estão ligadas à classe correspondente por uma linha a tracejado.

Ao concluir esta secção convém notar que ao criar a classe `compl` abandonámos a ideia de utilizar um dicionário para representar números complexos, optando por utilizar duas variáveis separadas, `self.r` e `self.i`, hipótese que tínhamos abandonado à partida no início deste capítulo (página 258). A razão da nossa decisão prende-se com o facto destas duas variáveis existirem dentro do objecto que corresponde a um número complexo e consequentemente, embora sejam variáveis separadas, elas estão mutuamente ligadas por existirem no mesmo objecto.

10.6. NOTAS FINAIS

279

`compl`

`self.r`

`self.i`

`p real`

`p imag`

`compl zero`

`imag puro`

`compl iguais`

`escreve`

`c1`

Figura 10.4: Representação da classe comp e suas instâncias.

10.6

Notas finais

Neste capítulo, apresentamos a metodologia dos tipos abstractos de informação, que permite separar as propriedades abstractas de um tipo do modo como ele é realizado numa linguagem de programação. Esta separação permite melhorar a tarefa de desenvolvimento de programas, a facilidade de leitura de um programa e torna os programas independentes da representação escolhida para os tipos de informação. Esta metodologia foi introduzida por [Liskof and Zilles, 1974] e posteriormente desenvolvida por [Liskof and Guttag, 1986].

De acordo com esta metodologia, sempre que criamos um novo tipo de informação devemos seguir os seguintes passos: (1) especificar as operações básicas para o tipo; (2) especificar as relações que as operações básicas têm de satisfazer; (3) escolher uma representação para o tipo; (4) realizar as operações básicas com base na representação escolhida.

Um tópico importante que foi discutido superficialmente neste capítulo tem a ver com garantir que as operações básicas que especificamos para o tipo na realidade caracterizam o tipo que estamos a criar. Esta questão é respondida fornecendo uma axiomatização para o tipo. Uma axiomatização corresponde a uma apresentação formal das propriedades do tipo. Exemplos de axiomatizações podem ser consultadas em [Hoare, 1972] e [Manna and Waldinger, 1985].

280

CAPÍTULO 10. ABSTRACC

~
s

AO DE DADOS

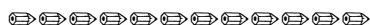
y *



 v1



v2



x

Figura 10.5: Exemplo de vectores.

10.7

Exercícios

1. Suponha que desejava criar o tipo `vector` em Python. Um vector num referencial cartesiano pode ser representado pelas coordenadas da sua extremidade (x, y) , estando a sua origem no ponto $(0, 0)$, ver Figura 10.5.

Podemos considerar as seguintes operações básicas para vectores:

- Construtor:

`vector : real  $\times$  real   $\rightarrow$  vector`

`vector(x, y)` tem como valor o vector cuja extremidade é o ponto (x, y) .

- Selectores:

`abcissa : vector   $\rightarrow$  real`

abscissa(v) tem como valor a abscissa da extremidade do vector v.

ordenada : vector $\rightarrow$ real

ordenada(v) tem como valor a ordenada da extremidade do vector v.

• Reconhecedores:

vector : universal $\rightarrow$ lógico

vector(arg) tem valor verdadeiro apenas se arg é um vector.

vector nulo : vector $\rightarrow$ lógico

vector nulo(v) tem valor verdadeiro apenas se v é o vector (0, 0).

• Teste:

vectores iguais : vector $\times$ vector $\rightarrow$ lógico

10.7. EXERCÍCIOS

281

vectores iguais(v1, v2) tem valor verdadeiro apenas se os vectores v1 e v2 são iguais.

(a) Defina uma representação para vectores utilizando tuplos.

(b) Escolha uma representação externa para vectores e escreva o transformador de saída.

(c) Implemente o tipo vector utilizando funções.

(d) Defina a classe vector.

2. Tendo em atenção as operações básicas sobre vectores da pergunta anterior, escreva funções em Python (quer usando a representação de funções, quer usando objectos) para:

(a) Somar dois vectores. A soma dos vectores representados pelos pontos (a, b) e (c, d) é dada pelo vector (a + c, b + d).

(b) Calcular o produto escalar de dois vectores. O produto escalar dos vectores representados pelos pontos (a, b) e (c, d) é dado pelo real a.c + b.d.

(c) Determinar se um vector é colinear com o eixo dos xx.

3. Suponha que quer representar o tempo, dividindo-o em horas e minutos.

(a) Especifique e implemente um tipo abstracto tempo. No seu tipo, o número de minutos está compreendido entre 0 e 59, e o número de horas apenas está limitado inferiormente a zero. Por exemplo 546:37

é um tempo válido.

(b) Com base no tipo da alínea anterior, escreva as seguintes funções:

- depois : tempo $\times$ tempo $\rightarrow$ lógico

depois(t1, t2) tem o valor verdadeiro, se t1 corresponder a um instante de tempo posterior a t2.

- num minutos : tempo $\rightarrow$ inteiro

num minutos(t) tem como valor o número de minutos entre o momento 0 horas e 0 minutos e o tempo t.

282

CAPÍTULO 10. ABSTRACC

~
,

AO DE DADOS

Capítulo 11

Programaç~

ao com objectos

And I haven't seen the two Messengers, either. They're both gone to the town. Just look along the road, and tell me if you can see either of them.

Lewis Carroll, Through the Looking Glass

No capítulo anterior introduzimos o conceito de objecto, uma entidade que contém um conjunto de variáveis internas e um conjunto de operações que manipulam essas variáveis. A abordagem que

fizemos a objectos correspondeu a criar objectos que correspondem a constantes do tipo complexo. Neste capítulo começamos por apresentar um tipo de informação correspondente a objectos cujas variáveis internas variam ao longo do tempo e analisamos qual a influência que este aspecto apresenta no seu comportamento.

A utilização de objectos em programação dá origem a um paradigma de programação conhecido por programação com objectos¹. Neste capítulo também abordamos alguns aspectos associados à programação com objectos.

¹Do inglês “object-oriented programming”. Em português, este tipo de programação também é conhecido por programação

orientada a objectos, programação

orientada por objectos, programação

orientada por objectos e programação

orientada aos objectos.

283

284

CAPÍTULO 11. PROGRAMAC

~
,

AO COM OBJECTOS

11.1

O tipo conta bancária

Dizemos que uma entidade tem estado se o seu comportamento é influenciado pela sua história. Vamos ilustrar o conceito de entidade com estado através da manipulação de uma conta bancária. Para qualquer conta bancária, a resposta à questão “posso levantar 20 euros?” depende da história dos depósitos e dos levantamentos efectuados nessa conta. A situação de uma conta bancária pode ser caracterizada por um dos seguintes aspectos: o saldo num dado instante, ou a sequência de todos os movimentos na conta desde a sua criação. A situação da conta bancária é caracterizada computacionalmente através do conceito de estado. O estado de uma entidade é caracterizado por uma ou mais variáveis, as variáveis de

estado, as quais mantêm informação sobre a história da entidade, de modo a poder determinar o comportamento da entidade.

Suponhamos que desejávamos modelar o comportamento de uma conta bancária.

Por uma questão de simplificação, caracterizamos uma conta bancária apenas pelo seu saldo, ignorando todos os outros aspectos, que sabemos estarem associados a contas bancárias, como o número, o titular, o banco, a agência, *etc.*

Para modelar o comportamento de uma conta bancária, vamos criar uma classe, chamada `conta`, cujo estado interno é definido pela variável `saldo` e está associada a funções que efectuam depósitos, levantamentos e consultas de saldo.

```
class conta:

    def __init__(self, quantia):

        self.saldo = quantia

    def deposito (self, quantia):

        self.saldo = self.saldo + quantia

        return self.saldo

    def levantamento (self, quantia):

        if self.saldo - quantia >= 0:

            self.saldo = self.saldo - quantia

            return self.saldo

        else:

            print('Saldo insuficiente')
```

11.1. O TIPO CONTA BANCÁRIA 285

```
def consulta (self):

    return self.saldo
```

Com esta classe podemos obter a seguinte interacção:

```
> > > conta_01 = conta(100)
```

```
> > > conta_01.deposito(50)
```

```
150
```

```
> > > conta_01.consulta()
```

```
150
```

```
> > > conta_01.levantamento(120)
```

```
30
```

```
> > > conta_01.levantamento(50)
```

Saldo insuficiente

```
> > > conta_02 = conta(250)
```

```
> > > conta_02.deposito(50)
```

```
300
```

```
> > > conta_02.consulta()
```

```
300
```

Uma conta bancária, tal como a que acabámos de definir, corresponde a uma entidade que não só possui um estado (o qual é caracterizado pelo saldo) mas também está associada a um conjunto de comportamentos (as funções que efectuam depósitos, levantamentos e consultas). A conta bancária evolui de modo diferente quando fazemos um depósito ou um levantamento.

Sabemos que em programação com objectos existe uma abstracção chamada classe. Uma classe corresponde a uma infinidade de objectos com as mesmas variáveis de estado e com o mesmo comportamento. Por exemplo, a conta bancária que definimos corresponde a uma classe de objectos cujo estado é definido pela variável saldo e cujo comportamento é definido pelas funções levantamento, deposito e consulta. As contas conta 01 e conta 02, definidas na nossa interacção, correspondem a instâncias da classe conta. Como já vimos no Capítulo 10, as classes e instâncias são representadas graficamente como se mostra na Figura 11.1.

Uma classe é um potencial gerador de instâncias, objectos cujas

variáveis de estado e comportamento são definidos pela classe. São as instâncias que contêm

286

CAPÍTULO 11. PROGRAMAC

~
,

AO COM OBJECTOS

conta

self.saldo

levantamento

deposito

consulta

conta-01

conta-02

Figura 11.1: Representação da classe conta e suas instâncias.

o estado interno e que reagem a solicitações – a classe apenas define como é caracterizado o estado e o comportamento das suas instâncias.

Ao trabalhar com objectos, é importante distinguir entre identidade e igualdade.

Considerando a classe conta, podemos definir as contas conta-01 e conta-02

do seguinte modo:

```
> > > conta_01 = conta(100)
```

```
> > > conta_02 = conta(100)
```

Será que os dois objectos conta 01 e conta 02 são iguais? Embora eles tenham sido definidos exactamente da mesma maneira, como instâncias da mesma classe, estes dois objectos têm estados distintos, como se demonstra pela seguinte interacção:

```
> > > conta_01 = conta(100)
```



```
> > > conta_02 = conta(100)
```

```
> > > conta_01.levantamento(25)
```

75

```
> > > conta_01.levantamento(35)
```

40

```
> > > conta_02.deposito(120)
```

220

```
> > > conta_01.consulta()
```

40

```
> > > conta_02.consulta()
```

11.2. CLASSES, SUBCLASSES E HERANÇAS

, A

287

220

Suponhamos agora que definíssemos:

```
> > > conta_01 = conta(100)
```

```
> > > conta_02 = conta_01
```

Repare-se que agora conta_01 e conta_02 são o mesmo objecto (correspondem na realidade a uma conta conjunta). Por serem o mesmo objecto, qualquer alteração a uma destas contas reflecte-se na outra:

```
> > > conta_01 = conta(100)
```

```
> > > conta_02 = conta_01
```

```
> > > conta_01.consulta()
```

100

```
> > > conta_02.levantamento(30)
```

```
> > > conta_01.consulta()
```

Classes, subclasses e herança

Uma das vantagens da programação com objectos é a possibilidade de construir entidades reutilizáveis, ou seja, componentes que podem ser usados em programas diferentes através da especialização para necessidades específicas.

Para ilustrar o aspecto da reutilização, consideremos, de novo, a classe *conta*.

Esta classe permite a criação de contas bancárias, com as quais podemos efectuar levantamentos e depósitos. Sabemos, no entanto, que qualquer banco oferece aos seus clientes a possibilidade de abrir diferentes tipos de contas, existem contas ordenado que permitem apresentar saldos negativos (até um certo limite), existem contas jovem que não permitem saldos negativos, mas que não impõem restrições mínimas para o saldo de abertura, mas, em contrapartida, não pagam juros, e assim sucessivamente. Quando um cliente abre uma nova conta, tipicamente define as características da conta desejada: “quero uma conta em que não exista pagamento adicional pela utilização de cheques”, “quero uma conta que ofereça 2% de juros por ano”, *etc.*

CAPÍTULO 11. PROGRAMAC

~
,

AO COM OBJECTOS

Todos estes tipos de contas são “contas bancárias”, mas apresentam características (ou especializações) que variam de conta para conta. A programação tradicional, aplicada ao caso das contas bancárias, levaria à criação de funções em que o tipo de conta teria de ser testado quando uma operação de abertura, de levantamento ou de depósito era efectuada. Em programação com objectos, a abordagem seguida corresponde à criação de classes que especializam outras

classes.

Comecemos por considerar uma classe de contas genéricas, conta gen, a qual sabe efectuar levantamentos, depósitos e consultas ao saldo. Esta classe não impõe qualquer restrição durante o levantamento.

A classe conta gen, correspondente a uma conta genérica, pode ser realizada através da seguinte definição:

```
class conta_gen:

    def __init__(self, quantia):

        self.saldo = quantia

    def deposito (self, quantia):

        self.saldo = self.saldo + quantia

    return self.saldo

    def levantamento (self, quantia):

        self.saldo = self.saldo - quantia

    return self.saldo

    def consulta(self):

    return self.saldo
```

Tendo definido a classe conta gen, podemos definir a instância conta gen 01 e efectuar as seguintes operações:

```
> > > conta_gen_01 = conta_gen(20)

> > > conta_gen_01.consulta()

20

> > > conta_gen_01.levantamento(300)
```

11.2. CLASSES, SUBCLASSES E HERANÇAS

, A

```
> > > conta_gen_01.deposito(400)
```

120

Suponhamos agora que estávamos interessados em criar os seguintes tipos de contas:

- **conta ordenado:** esta conta está associada à transferência mensal do ordenado do seu titular e está sujeita às seguintes restrições:
 - a sua abertura exige um saldo mínimo igual ao valor do ordenado a ser transferido para a conta.
 - permite saldos negativos, até um montante igual ao ordenado.
- **conta jovem:** conta feita especialmente para jovens e que está sujeita às seguintes restrições:
 - a sua abertura não exige um saldo mínimo.
 - não permite saldos negativos.

Para criar as classes `conta ordenado` e `conta jovem`, podemos pensar em duplicar o código associado à classe `conta gen`, fazendo as adaptações necessárias nos respectivos métodos. No entanto, esta é uma má prática de programação, pois não só aumenta desnecessariamente a quantidade de código, mas também porque qualquer alteração realizada sobre a classe `conta gen` não se propaga automaticamente às classes `conta ordenado` e `conta jovem`, as quais são contas.

Para evitar estes inconvenientes, em programação com objectos existe o conceito de subclasse. Diz-se que uma classe é uma subclasse de outra classe, se a primeira corresponder a uma especialização da segunda. Ou seja, o comportamento da subclasse corresponde ao comportamento da superclasse, excepto no caso em que comportamento específico está indicado para a subclasse. Diz-se que a subclasse herda o comportamento da superclasse, excepto quando este é explicitamente alterado na subclasse.

É na definição de subclasses que temos que considerar a parte opcional da

◆ definição de objecto ◆ apresentada na página 273, e reproduzida

aqui para facilitar a leitura:

290

CAPÍTULO 11. PROGRAMAC

~
s

AO COM OBJECTOS

◆definiç~ao de objecto◆ :: = class ◆nome◆ {(◆nome◆)}: CR

TAB ◆definiç~ao de método◆ +

Ao considerar uma definiç~ao de classe da forma class◆nome1◆ (◆nome2◆), estamos a definir a classe ◆nome1◆ como uma subclasse de ◆nome2◆. Neste caso, a classe ◆nome1◆ automaticamente “herda” todas as variáveis e métodos definidos na classe ◆nome2◆.

Por exemplo, definindo a classe conta ordenado como class conta ordenado (conta gen),

se nada for dito em contrário, a classe conta ordenado passa automaticamente a ter a variável self.saldo e os métodos init , deposito, levantamento e consulta definidos na classe conta gen. Se, associado à classe conta ordenado, for definido um método com o mesmo nome de um método da classe conta gen, este método, na classe conta ordenado, sobrep~oe-se ao método homónimo da classe conta gen. Se forem definidos novos métodos na classe conta ordenado, estes pertencem apenas a essa classe, não existindo na classe conta gen.

Em resumo, as classes podem ter subclasses que correspondem a especializaç~oes dos seus elementos. As subclasses herdam as variáveis de estado e os métodos das superclasses, salvo indicaç~ao em contrário.

A classe conta ordenado pode ser definida do seguinte modo: class conta_ordenado (conta_gen):

```
def __init__ (self, quantia, ordenado):
```

```
if quantia >= ordenado:
```

```
self.saldo = quantia
```

```
self.ordenado = ordenado
```

```
else:
```

```
print('O saldo deve ser maior que o ordenado') def levantamento (self,
quantia):
```

```
if quantia <= self.saldo + self.ordenado: self.saldo = self.saldo -
quantia
```

```
return self.saldo
```

```
else:
```

11.2. CLASSES, SUBCLASSES E HERANÇAS

, A

291

```
print('Saldo insuficiente')
```

Nesta classe, o método `init` é redefinido com um argumento adicional correspondente ao `ordenado`, o qual é guardado numa variável própria da classe `conta ordenado`. O método `levantamento` é também redefinido, de modo a verificar que o saldo nunca desce abaixo do simétrico do `ordenado`. Os métodos `deposito` e `consulta` são herdados da classe `conta gen`.

Analogamente, a classe `conta jovem` é definida como uma subclasse de `conta gen`, redefinindo apenas o método `levantamento`.

```
class conta_jovem (conta_gen):
```

```
def levantamento (self, quantia):
```

```
if quantia <= self.saldo:
```

```
self.saldo = self.saldo - quantia
```

```
return self.saldo
```

```
else:
```

```
print('Saldo insuficiente')
```

Estas classes permitem-nos efectuar a seguinte interacção:

```
>>> conta_ord_01 = conta_ordenado(300, 500) O saldo deve ser maior que o ordenado
```

```
>>> conta_ord_01 = conta_ordenado(800, 500)
```

```
>>> conta_ord_01.levantamento(500)
```

```
300
```

```
>>> conta_ord_01.levantamento(500)
```

```
-200
```

```
>>> conta_jov_01 = conta_jovem(50)
```

```
>>> conta_jov_01.consulta()
```

```
50
```

```
>>> conta_jov_01.levantamento(100)
```

Saldo insuficiente

As classes correspondentes aos diferentes tipos de contas estão relacionadas entre si, podendo ser organizadas numa hierarquia. Nesta hierarquia, ao nível mais alto está a classe conta gen, com duas subclasses, conta ordenado e

292

CAPÍTULO 11. PROGRAMAC

~
,

AO COM OBJECTOS

conta gen

self.saldo

levantamento

deposito

consulta



conta ordenado

conta jovem

self.ordenado

levantamento

levantamento

conta-ord-01

conta-jov-01

Figura 11.2: Hierarquia de contas e respectivas instâncias.

conta jovem. Na Figura 11.2 mostramos esta hierarquia, bem como as instâncias conta ord 01 e conta jov 01.

A importância da existência de várias classes e instâncias organizadas numa hierarquia, que estabelece relações entre classes e suas subclasses e instâncias, provém de uma forma de propagação associada, designada por herança. A herança consiste na transmissão das características duma classe às suas subclasses e instâncias. Isto significa, por exemplo, que todos os métodos associados à classe conta gen são herdados por todas as subclasses e instâncias dessa classe, excepto se forem explicitamente alterados numa das subclasses.

Suponhamos agora que desejamos criar uma conta jovem que está protegida por um código de segurança, normalmente conhecido por PIN2. Esta nova conta, sendo uma conta jovem, irá herdar as características desta classe. A nova classe que vamos definir, conta jovem com pin terá um estado interno constituído por três variáveis, uma delas corresponde ao código de acesso, pin, a outra 2Do inglês “Personal Identification Number”.

11.2. CLASSES, SUBCLASSES E HERANCA

, A

293

será uma conta jovem, a qual só é acedida após o fornecimento do código de acesso correcto, finalmente, a terceira variável, contador, regista o número de tentativas falhadas de acesso à conta, bloqueando o acesso à conta assim que este número atingir o valor 3. Deste modo, o método que cria contas da classe conta jovem com pin é definido

por:

```
def __init__(self, quantia, codigo):  
  
    self.conta = conta_jovem(quantia)  
  
    self.pin = codigo  
  
    self.contador = 0
```

Para aceder a uma conta do tipo conta jovem com pin utilizamos o método `accede`, o qual, sempre que é fornecido um PIN errado aumenta em uma unidade o valor do contador de tentativas de acesso incorrectas. Ao ser efectuado um acesso com o PIN correcto, os valor do contador de tentativas incorrectas volta a ser considerado zero. O método `accede` devolve `True` se é efectuado um acesso com o PIN correcto e devolve `False` em caso contrário.

```
def accede(self, pin):  
  
    if self.contador == 3:  
  
        print('Conta bloqueada')  
  
        return False  
  
    elif pin == self.pin:  
  
        self.contador = 0  
  
        return True  
  
    else:  
  
        self.contador = self.contador + 1  
  
        if self.contador == 3:  
  
            print('Conta bloqueada')  
  
            else:  
  
                print('PIN incorrecto')  
  
                print('Tem mais ', 3-self.contador, ' tentativas') return False
```

Os métodos `levantamento`, `deposito` e `consulta` utilizam o método `accede` para verificar a correcção do PIN e, em caso de este acesso

estar correcto, utilizam o método correspondente da conta conta jovem para aceder à conta

294

CAPÍTULO 11. PROGRAMAC

~
,

AO COM OBJECTOS

`self.conta`. Note-se que `self.conta` é a instância da classe `conta jovem` que está armazenada na instância da classe `conta jovem` com `pin` que está a ser manipulada.

Finalmente, a classe `conta jovem` com `pin` apresenta um método que não existe na classe `conta jovem` e que corresponde à acção de alteração do PIN da conta.

A classe `conta jovem` com `pin` é definida do seguinte modo: `class conta_jovem_com_pin (conta_jovem):`

```
def __init__(self, quantia, codigo):  
  
    self.conta = conta_jovem (quantia)  
  
    self.pin = codigo  
  
    self.contador = 0  
  
    def levantamento (self, quantia, pin):  
  
        if self.acede(pin):  
  
            return self.conta.levantamento(quantia)  
  
        def deposito (self, quantia, pin):  
  
            if self.acede(pin):  
  
                return self.conta.deposito(quantia)  
  
        def consulta (self, pin):  
  
            if self.acede(pin):  
  
                return self.conta.consulta()
```

```

def altera_codigo (self, pin):

    if self.acede(pin):

        novopin = input('Introduza o novo PIN\n--> ') print('Para verificaç~
ao')

        verifica = input('Volte a introduzir o novo PIN\n--> ') if novopin
        == verifica:

            self.pin = novopin

            print('PIN alterado')

        else:

            print('Operaç~
ao sem sucesso')

```

11.2. CLASSES, SUBCLASSES E HERANC

, A

295

```

def acede (self, pin):

    if self.contador == 3:

        print('Conta bloqueada')

        return False

    elif pin == self.pin:

        self.contador = 0

        return True

    else:

        self.contador = self.contador + 1

    if self.contador == 3:

        print('Conta bloqueada')

```

else:

```
print('PIN incorrecto')
```

```
print('Tem mais ', 3-self.contador, ' tentativas') return False
```

Com esta classe, podemos originar a seguinte interacção:

```
> > > conta_jcp_01 = conta_jovem_com_pin(120, '1234')
```

```
> > > conta_jcp_01.levantamento(30, 'abcd') PIN incorrecto
```

Tem mais

2

tentativas

```
> > > conta_jcp_01.levantamento(30, 'ABCD') PIN incorrecto
```

Tem mais

1

tentativas

```
> > > conta_jcp_01.levantamento(30, '1234') 90
```

```
> > > conta_jcp_01.altera_codigo('abcd') PIN incorrecto
```

Tem mais

2

tentativas

```
> > > conta_jcp_01.altera_codigo('1234') Introduza o novo PIN
```

```
--> abcd
```

Para verificaç~

ao

Volte a introduzir o novo PIN

```
--> abcd
```

CAPÍTULO 11. PROGRAMAC

~
,

AO COM OBJECTOS

PIN alterado

> > > conta_jcp_01.levantamento(30, 'abcd') 60

> > > conta_jcp_01.levantamento(30, '1234') PIN incorrecto

Tem mais

2

tentativas

> > > conta_jcp_01.levantamento(30, '1234') PIN incorrecto

Tem mais

1

tentativas

> > > conta_jcp_01.levantamento(30, '1234') Conta bloqueada

11.3

Objectos em Python

O Python é uma linguagem baseada em objectos. Embora a nossa utilização do Python tenha fundamentalmente correspondido à utilização da sua faceta imperativa, na realidade, todos os tipos de informação embutidos em Python correspondem a classes.

Recorrendo à função `type`, introduzida na página 277, podemos obter a seguinte interacção que revela que os tipos embutidos com que temos trabalhado não são mais do que objectos pertencentes a determinadas classes:

> > > `type(2)`

<class 'int'>

> > > `type(True)`

```
<class 'bool'>
```

```
> > > type(3.5)
```

```
<class 'float'>
```

```
> > > type('abc')
```

```
<class 'str'>
```

```
> > > type([1, 2, 3])
```

```
<class 'list'>
```

Recordemos a função que lê uma linha de um ficheiro, `◆fich◆.readline()`, apresentada na Secção 8.2. Dissemos na altura que o nome desta função podia ser tratado como um nome composto. O que na realidade se passa é que ao abrirmos

11.4. POLIMORFISMO

297

um ficheiro, estamos a criar um objecto, objecto esse em que um dos métodos tem o nome de `readline`.

11.4

Polimorfismo

Já vimos que muitas das operações do Python são operações sobrecarregadas, ou seja, aplicam-se a vários tipos de informação. Por exemplo, a operação `+`

pode ser aplicada a inteiros, a reais, a tuplos, a cadeias de caracteres, a listas e a dicionários.

Diz-se que uma operação é polimórfica, ou que apresenta a propriedade do polimorfismo³, quando é possível definir funções diferentes que usam a mesma operação para lidar com tipos de dados diferentes. Consideremos a operação de adição, representada em Python por `+`. Esta operação aplica-se tanto a inteiros como a reais. Recorde-se da página 41, que dissemos que internamente ao Python existiam duas operações, que representámos por `+Z` e `+R`, que, respectivamente, são invocadas quando os argumentos da operação `+` são números inteiros ou são números reais. Esta capacidade do Python de associar a mesma representação externa de

uma operação, +, a diferentes operações internas corresponde a uma faceta do polimorfismo⁴. Esta faceta do polimorfismo dá origem a operações sobrecarregadas.

Após definirmos o tipo correspondente a números complexos, no Capítulo 10, estaremos certamente interessados em escrever funções que executem operações aritméticas sobre complexos. Por exemplo, a soma de complexos pode ser definida por:

```
def soma_complexos (c1, c2):  
  
    r = c1.p_real() + c2.p_real()  
  
    i = c1.p_imag() + c2.p_imag()  
  
    return compl(r, i)
```

com a qual podemos gerar a interacção:

```
> > > c1 = compl(9, 6)
```

3A palavra “polimorfismo” é derivada do grego e significa apresentar múltiplas formas.

4O tipo de polimorfismo que definimos nesta secção corresponde ao conceito inglês de

“subtype polymorphism”.

298

CAPÍTULO 11. PROGRAMAC

~
,

AO COM OBJECTOS

```
> > > c2 = compl(7, 6)  
  
> > > c3 = soma_complexos(c1, c2)  
  
> > > c3.escreve()
```

16 + 12 i

No entanto, quando trabalhamos com números complexos em matemática usamos o símbolo da operação de adição, +, para

representar a soma de números complexos. Através da propriedade do polimorfismo, o Python permite especificar que a operação `+` também pode ser aplicada a números complexos e instruir o computador em como aplicar a operação `+` a números complexos.

Sabemos que em Python, todos os tipos embutidos correspondem a classes.

Um dos métodos que existe numa classe tem o nome de `add`, recebendo

dois argumentos, `self` e outro elemento do mesmo tipo. A sua utilização é semelhante à da função `compl` iguais que apresentámos na página 275, como o mostra a seguinte interacção:

```
>>> a = 5
```

```
>>> a._add_(2)
```

```
7
```

A invocação deste método pode ser feita através da representação externa `+`, ou seja, sempre que o Python encontra a operação `+`, invoca automaticamente o método `add`, aplicado às instâncias envolvidas. Assim, se associado à classe `compl`, definirmos o método

```
def _add_(self, outro):
```

```
    r = self.p_real() + outro.p_real()
```

```
    i = self.p_imag() + outro.p_imag()
```

```
    return compl(r, i)
```

podemos originar a interacção

```
>>> c1 = compl(2, 4)
```

```
>>> c2 = compl(5, 10)
```

```
>>> c3 = c1 + c2
```

```
>>> c3.escreve()
```

```
7 + 14 i
```

11.4. POLIMORFISMO

Ou seja, usando o polimorfismo, criámos uma nova forma da operação `+` que sabe somar complexos. A “interface” da operação para somar complexos passa a ser o operador `+`, sendo este operador transformado na operação de somar complexos quando os seus argumentos são números complexos.

De um modo análogo, existem métodos embutidos, com os nomes sub

`mul`

e `truediv` que estão associados às representações das operações `-`,

`*` e `/`. O método

`eq`

está associado à operação `==`. Existe também um método, `repr` que transforma a representação interna de uma instância da classe numa cadeia de caracteres que corresponde à sua representação externa.

Esta representação externa é usada directamente pela função `print`.

Sabendo como definir em Python operações aritméticas e a representação externa, podemos modificar a classe `compl` com os seguintes métodos. Repare-se que com o método `eq` deixamos de precisar do método `compl` iguais e que com o método `repr` deixamos de precisar do método `escreve` para gerar a representação externa de complexos5:

```
class compl:
```

```
    def __init__(self, real, imag):
```

```
        if isinstance(real, (int, float)) and \
```

```
            isinstance(imag, (int, float)):
```

```
            self.r = real
```

```
            self.i = imag
```

```
        else:
```

```
            raise ValueError('complexo: argumento errado')
    def p_real(self):
```

```
return self.r
```

```
def p_imag (self):
```

```
return self.i
```

```
def zero (self):
```

```
return self.r == 0 and self.i == 0
```

5No método repr utilizamos a função embutida str que recebe uma constante de qualquer tipo e tem como valor a cadeia de caracteres correspondente a essa constante. Por exemplo, str(3) tem o valor '3' e str([2, 3, 4]) tem o valor '[2, 3, 4]'.

300

CAPÍTULO 11. PROGRAMAC

~
,

AO COM OBJECTOS

```
def __eq__ (self, outro):
```

```
return self.r == outro.p_real() and \
```

```
self.i == outro.p_imag()
```

```
def __add__ (self, outro):
```

```
r = self.p_real() + outro.p_real()
```

```
i = self.p_imag() + outro.p_imag()
```

```
return compl(r, i)
```

```
def __sub__ (self, outro):
```

```
r = self.p_real() - outro.p_real()
```

```
i = self.p_imag() - outro.p_imag()
```

```
return compl(r, i)
```

```
def __mul__ (self, outro):
```

```
r = self.p_real() * outro.p_real() - \
```

```

self.p_imag() * outro.p_imag()

i = self.p_real() * outro.p_imag() + \
self.p_imag() * outro.p_real()

return compl(r, i)

def __truediv__(self, outro):
    try:
        den = outro.p_real() * outro.p_real() + \
        outro.p_imag() * outro.p_imag()
        r = (self.p_real() * outro.p_real() + \
        self.p_imag() * outro.p_imag()) / den
        i = (self.p_imag() * outro.p_real() - \
        self.p_real() * outro.p_imag()) / den
        return compl(r, i)
    except ZeroDivisionError:
        print('complexo: divis~
        ao por zero')

def __repr__(self):
    if self.p_imag() >= 0:
        return str(self.p_real()) + '+' + \

```

11.5. NOTAS FINAIS

301

```

str(self.p_imag()) + 'i'
else:
    return str(self.p_real()) + '-' + \

```

```
str(abs(self.p_imag())) + 'i'
```

Podendo agora gerar a seguinte interacção:

```
>>> c1 = compl(2, 5)
```

```
>>> c1
```

2+5i

```
>>> c2 = compl(-9, -7)
```

```
>>> c2
```

-9-7i

```
>>> c1 + c2
```

-7-2i

```
>>> c1 * c2
```

17-59i

```
>>> c3 = compl(0, 0)
```

```
>>> c1 / c3
```

complexo: divis~

ao por zero

```
>>> c1 == c2
```

False

```
>>> c4 = compl(2, 5)
```

```
>>> c1 == c4
```

True

11.5

Notas finais

Neste capítulo apresentámos um estilo de programação, conhecido como programação com objectos, que é centrado em objectos,

entidades que possuem um estado interno e reagem a mensagens. Os objectos correspondem a entidades, tais como contas bancárias, livros, estudantes, *etc.* O conceito de objecto agrupa as funções que manipulam dados (os métodos) com os dados que representam o estado do objecto.

O conceito de objecto foi introduzido com a linguagem SIMULA6, foi vulgarizado [Dahl and Nygaard, 1967].

302

CAPÍTULO 11. PROGRAMAC

~
,

AO COM OBJECTOS

zado em 1984 pelo sistema operativo do Macintosh, foi adoptado pelo sistema operativo Windows, uma década mais tarde, e está hoje na base de várias linguagens de programação, por exemplo, o C++, o CLOS, o Java e o Python.

11.6

Exercícios

11.1 Defina uma classe em Python, chamada estacionamento, que simula o funcionamento de um parque de estacionamento. A classe estacionamento recebe um inteiro que determina a lotação do parque e devolve um objecto com os seguintes métodos: `entra()`, corresponde à entrada de um carro; `sai()`, corresponde à saída de um carro; `lugares()` indica o número de lugares livres no estacionamento. Por exemplo,

```
>>> ist = estacionamento(20)
```

```
>>> ist.lugares()
```

20

```
>>> ist.entra()
```

```
>>> ist.entra()
```

```
>>> ist.entra()
```

```
>>> ist.entra()
```

```
> > > ist.sai()
```

```
> > > ist.lugares()
```

17

11.2 Defina uma classe que corresponde a uma urna de uma votação. A sua classe deve receber a lista dos possíveis candidatos e manter como estado interno o número de votos em cada candidato. Esta classe pode receber um voto num dos possíveis candidatos, aumentando o número de votos nesse candidato em um. Deve também permitir apresentar os resultados da votação.

Capítulo 12

O desenvolvimento de

programas

‘First, the fish must be caught,’

That is easy: a baby, I think could have caught it.

‘Next, the fish must be bought,’

That is easy: a penny, I think, would have bought it.

‘Now, cook me the fish!’

That is easy, and will not take more than a minute.

‘Let it lie in a dish!’

That is easy, because it already is in it.

Lewis Carroll, Through the Looking Glass

A finalidade deste capítulo é a apresentação sumária das várias fases por que passa o desenvolvimento de um programa, fornecendo uma visão global da actividade de programação.

A expressão “desenvolvimento de um programa” é frequentemente considerada como sinónimo de programação, ou de codificação, isto é, a escrita de instruções utilizando uma linguagem de programação. Contudo, bastante trabalho preparatório deve anteceder a programação de qualquer solução potencial para o problema que se pretende resolver. Este trabalho preparatório é constituído por fases

como a definição exacta do que se pretende fazer, removendo ambiguidades e incertezas que possam estar contidas nos objectivos a atingir, a decisão do processo a utilizar para a solução do problema e o delineamento da solução utilizando uma linguagem adequada. Se este trabalho preparatório for bem feito, a 303

304

CAPÍTULO 12. O DESENVOLVIMENTO DE PROGRAMAS

la ✱

rm

Fo

Projecto

grande e

complexo



toc



je



o

pr



de



o



stãe



G



la

Pequeno

rm

projecto

o

Inf



Informal

Formal

Método de desenvolvimento

Figura 12.1: O problema da dimensã do programa.

fase de programaçã, que parece a mais importante para muitas pessoas, torna-se relativamente fácil e pouco criativa. Tal como se despendeu muito trabalho antes de começar a programar, muito trabalho terá de ser feito desde a fase de programaçã até o programa estar completo. Terã de se detectar e corrigir todos os erros, testar exaustivamente o programa e consolidar a documentaçã.

Mesmo depois de o programa estar completamente terminado, existe trabalho a fazer relativamente à manutençã do programa.

O ponto fundamental nesta discussã é que o desenvolvimento de um programa é uma actividade complexa, constituída por várias fases individualizadas, sendo todas elas importantes, e cada uma delas

contribuindo para a solução do problema. É evidente que, quanto mais complexo for o programa, mais complicada é a actividade do seu desenvolvimento.

Desde os anos 60 que a comunidade informática tem dedicado um grande esforço à caracterização e regulamentação da actividade de desenvolvimento de programas complexos. Este trabalho deu origem a uma subdisciplina da informática, a engenharia da programação¹ que estuda as metodologias para o desenvolvi-1Do inglês “software engineering”.

305

Análise do problema



Desenvolvimento da soluç~

ao



Programaç~

ao da soluç~

ao



Testes



Manutenç~

ao

Figura 12.2: Modelo de cascata.

mento de programas. A finalidade da engenharia da programação é a de criar metodologias para desenvolver programas de qualidade a baixo custo. Nesta disciplina faz-se a distinção clara entre o conceito de programa (conjunto de instruções escritas numa linguagem de programação) e sistema computacional, entendido como “software”. A sociedade americana IEEE, “Institute of Electrical and Electronics

Engineers”, definiu o termo sistema computacional como sendo “uma colecção de programas, funções, regras, documentação e dados associados” para transmitir de modo inequívoco que a actividade de programação não se limita à produção de código para ser executado por um computador, mas que inclui toda a documentação e dados associados a esse código.

Existem diferentes metodologias para diferentes tipos de problemas a resolver e diferentes dimensões do programa final. Um dos problemas associados ao desenvolvimento de programas é a dimensão do produto final. Os métodos utilizados para o desenvolvimento de pequenos programas não podem ser adaptados directamente a programas grandes e complexos (Figura 12.1). Quanto maior for a complexidade do programa e a dimensão da equipa associada ao seu desenvolvimento, maior é o nível de formalização necessário para os métodos de desenvolvimento e para a gestão do projecto associado ao seu desenvolvimento.

Uma das metodologias mais usadas no desenvolvimento de programas é baseada no modelo da cascata². De acordo com o modelo da cascata, o desenvolvimento de um programa passa por cinco fases distintas que, embora sejam executadas sequencialmente, estão intimamente interligadas (Figura 12.2): a análise do pro-2Do inglês “waterfall method” (ver [Boehm, 1981]).

306

CAPÍTULO 12. O DESENVOLVIMENTO DE PROGRAMAS

blema, o desenvolvimento da solução, a programação da solução, os testes e a manutenção. Em todas estas fases é produzida documentação que descreve as decisões tomadas e que serve de apoio ao desenvolvimento das fases subsequentes.

Durante o desenvolvimento de um programa é bom ter sempre em mente a chamada Lei de Murphy:

1. Tudo é mais difícil do que parece.
2. Tudo demora mais tempo do que pensamos.
3. Se algo puder correr mal, irá correr mal, no pior dos momentos possíveis.

A análise do problema

Um programa é desenvolvido para satisfazer uma necessidade reconhecida por um utilizador ou por um conjunto de utilizadores. Neste capítulo, o utilizador (ou utilizadores) é designado por “cliente”. As necessidades do cliente são, de um modo geral, apresentadas com lacunas, imperfeições, ambiguidades e até, por vezes, contradições.

Durante a fase da análise do problema, o programador (ou, de um modo mais preciso, o analista – a pessoa que analisa o problema) estuda o problema, juntamente com o cliente, para determinar exactamente o que tem de ser feito, mesmo antes de pensar como os objectivos vão ser atingidos. Esta fase parece ser de tal maneira óbvia que muitos programadores a ignoram completamente, não perdendo tempo a tentar compreender o problema que se propõem resolver. Como resultado da não consideração desta fase, começam a desenvolver um programa mal concebido, o qual pode representar uma solução incorrecta do problema.

Esta fase processa-se antes de se começar a pensar na solução do problema, mais concretamente, em como resolver o problema. Pretende-se determinar claramente quais as especificações do problema e saber exactamente quais os objectivos a atingir. Esta fase envolve duas entidades com características diferentes, o programador e o cliente. De um modo geral, o programador não tem conhecimento sobre o domínio de aplicação do cliente e o cliente não domina os

12.2. O DESENVOLVIMENTO DA SOLUÇÃO

~
,

AO

307

aspectos técnicos associados à programação. Este aspecto cria um problema de comunicação que tem de ser resolvido durante a análise do problema.

O trabalho desenvolvido nesta fase inclui um conjunto de interacções entre o programador e o cliente, na qual, progressivamente, ambas as partes aumentam a sua compreensão sobre o que tem de ser feito. Estas interacções são baseadas num documento evolutivo que vai descrevendo, cada vez de modo mais detalhado, quais as características do trabalho a desenvolver.

O resultado desta fase é a criação de um documento – o documento de análise dos requisitos – especificando claramente, do ponto de vista informático (mas de modo que sejam perfeitamente entendidos pelo cliente), o que faz o programa, estudo das alternativas para o desenvolvimento do programa e riscos envolvidos no desenvolvimento. Para além deste documento, são também resultados da fase de análise do problema um planeamento pormenorizado para o desenvolvimento com o faseamento das fases seguintes, os custos estimados, *etc.*

O documento de análise de requisitos (conhecido frequentemente por SRS3) embora não especifique como é que o programa vai abordar a solução do problema, tem duas finalidades importantes. Por um lado, para o cliente, serve de garantia escrita do que vai ser feito. Por outro lado, para o programador, serve como definição dos objectivos a atingir.

12.2

O desenvolvimento da solução

ao

Uma vez sabido o que deve ser feito, durante o desenvolvimento da solução determina-se como deve ser feito. Esta é uma fase predominantemente criativa, em que se desenvolve um algoritmo que constitui a solução do problema a resolver.

O desenvolvimento deste algoritmo deve ser feito sem ligação a uma linguagem de programação particular, pensando apenas em termos das operações e dos tipos de informação que vão ser necessários. Normalmente, os algoritmos são desenvolvidos utilizando linguagens semelhantes às linguagens de programação nas quais se admitem certas descrições em língua natural. Pretende-se, deste modo, descrever rigorosamente como vai ser resolvido o problema sem se entrar, no entanto, nos pormenores inerentes a uma linguagem de programação.

3Do inglês, “System Requirements Specifications”.

CAPÍTULO 12. O DESENVOLVIMENTO DE PROGRAMAS

A ideia chave a utilizar durante esta fase é a abstracção. Em qualquer instante deve separar-se o problema que está a ser abordado dos

pormenores irrelevantes para a sua solução.

As metodologias a seguir durante esta fase são o desenvolvimento do topo para a base e a refinação por passos⁵, as quais têm como finalidade a diminuição da complexidade do problema a resolver, originando também algoritmos mais legíveis e fáceis de compreender. Segundo estas metodologias, o primeiro passo para a solução de um problema consiste em identificar os principais subproblemas que constituem o problema a resolver, e em determinar qual a relação entre esses subproblemas. Depois de concluída esta primeira fase, desenvolve-se uma primeira aproximação do algoritmo, aproximando essa que é definida em termos dos subproblemas identificados e das relações entre eles. Depois deve repetir-se sucessivamente este processo para cada um dos subproblemas. Quando se encontrar um subproblema cuja solução é trivial, deve-se então escrever o algoritmo para esse subproblema.

Esta metodologia para o desenvolvimento de um algoritmo tem duas vantagens: o controle da complexidade e a modularidade da solução. Quanto ao controle da complexidade, devemos notar que em cada instante durante o desenvolvimento do algoritmo estamos a tentar resolver um subproblema único, sem nos preocuparmos com os pormenores da solução, mas apenas com os seus aspectos fundamentais. Quanto à modularidade da solução, o algoritmo resultante será constituído por módulos, cada módulo correspondente a um subproblema cuja função é perfeitamente definida.

Seguindo esta metodologia, obtém-se um algoritmo que é fácil de compreender, de modificar e de corrigir. O algoritmo é fácil de compreender, pois é expresso em termos das divisões naturais do problema a resolver. Se pretendermos alterar qualquer aspecto do algoritmo, podemos determinar facilmente qual o módulo do algoritmo (ou seja, o subproblema) que é afectado pela alteração, e só teremos de considerar esse módulo durante o processo de modificação. Os algoritmos tornam-se assim mais fáceis de modificar. Analogamente, se detectarmos um erro na concepção do algoritmo, apenas teremos de considerar o subproblema em que o erro foi detectado.

O resultado desta fase é um documento, o documento de concepção (conhecido 4Do inglês, “top down design”).

5Do inglês, “stepwise refinement”.

12.3. A PROGRAMAC

frequentemente por SDD6, em que se descreve pormenorizadamente a soluç~ao do problema, as decis~oes que foram tomadas para o seu desenvolvimento e as decis~oes que têm de ser adiadas para a fase da programaç~ao da soluç~ao. O documento de concepç~ao está normalmente organizado em dois subdocumentos, a concepç~ao global e a concepç~ao pormenorizada. O documento de concepç~ao global identifica os módulos principais do sistema, as suas especificaç~oes de alto n~ivel, o modo de interacç~ao entre os módulos, os dados que recebem e os resultados que produzem. O documento de concepç~ao pormenorizada especifica o algoritmo utilizado por cada um dos módulos e os tipos de informaç~ao que estes manipulam.

A fase de desenvolvimento da soluç~ao termina com uma verificaç~ao formal dos documentos produzidos.

12.3

A programaç~

ao da soluç~

ao

The sooner you start coding your program the longer it is going to take.

[Ledgard, 1975]

Antes de iniciarmos esta secç~ao, será importante ponderarmos a citaç~ao de Henri Ledgard: “Quanto mais cedo começares a escrever o teu programa mais tempo demorarás”. Com esta frase, Ledgard pretende dizer que o desenvolvimento de um programa sem um período prévio de meditaç~ao e planeamento, em relaç~ao ao que deve ser feito e como fazê-lo, leva a situaç~oes caóticas, que para serem corrigidas requerem mais tempo do que o tempo despendido num planeamento cuidado do algoritmo.

Só depois de termos definido claramente o problema a resolver e de termos desenvolvido cuidadosamente um algoritmo para a sua solução, poderemos iniciar a fase de programação, ou seja, a escrita do algoritmo desenvolvido, recorrendo a uma linguagem de programação.

O primeiro problema a resolver, nesta fase, será a escolha da linguagem de programação a utilizar. A escolha da linguagem de programação é ditada por duas considerações fundamentais:

6Do inglês “Software Design Description”.

310

CAPÍTULO 12. O DESENVOLVIMENTO DE PROGRAMAS

1. As linguagens existentes (ou potencialmente existentes) no computador que vai ser utilizado. De facto, esta é uma limitação essencial. Apenas poderemos utilizar as linguagens que se encontram à nossa disposição.

2. A natureza do problema a resolver. Algumas linguagens são mais adequadas à resolução de problemas envolvendo fundamentalmente cálculos numéricos, ao passo que outras linguagens são mais adequadas à resolução de problemas envolvendo manipulações simbólicas. Assim, tendo a possibilidade de escolha entre várias linguagens, não fará sentido, por exemplo, a utilização de uma linguagem de carácter numérico para a solução de um problema de carácter simbólico.

Uma vez decidida qual a linguagem de programação a utilizar, e tendo já uma descrição do algoritmo, a geração das instruções do programa é relativamente fácil. O programador terá de decidir como representar os tipos de informação necessários e escrever as respectivas operações. Em seguida, traduzirá as instruções do seu algoritmo para instruções escritas na linguagem de programação a utilizar. O objectivo desta fase é a concretização dos documentos de concepção.

O resultado desta fase é um programa escrito na linguagem escolhida, com comentários que descrevem o funcionamento das funções e os tipos de informação utilizados, bem como um documento que complementa a descrição do algoritmo produzido na fase anterior e que descreve as decisões tomadas quanto à representação dos tipos de informação.

Juntamente com estes documentos s~ao apresentados os resultados dos testes que foram efectuados pelo programador para cada um dos m~odulos que comp~oem o sistema (os chamados testes de m~odulo ou testes unit~arios). Paralelamente, nesta fase desenvolve-se a documenta~ao de utiliza~ao do programa.

12.3.1

A depura~ao

ao

Anyone who believes his or her program will run correctly the first time is either a fool, an optimist, or a novice programmer.

[Schneider et al., 1978]

Na fase de depura~ao (do verbo depurar, tornar puro, em ingl~es, conhecida

12.3. A PROGRAMAC

~
,

AO DA SOLUC

~
,

AO

311

por “debugging”7) o programador detecta, localiza e corrige os erros existentes no programa desenvolvido. Estes erros fazem com que o programa produza resultados incorrectos ou n~ao produza quaisquer resultados. A fase de depura~ao pode ser a fase mais demorada no desenvolvimento de um programa.

Existem muitas raz~oes para justificar a grande quantidade de tempo e esfor~ao despendidos normalmente na fase de depura~ao, mas duas s~ao de import~ancia primordial. Em primeiro lugar, a facilidade de detec~ao de erros num programa est~a directamente relacionada com a clareza da estrutura do programa. A utiliza~ao de abstra~ao diminui a complexidade de um programa, facilitando a sua depura~ao. Em segundo lugar, as t~ecnicas de depura~ao n~ao s~ao normalmente ensinadas do mesmo modo sistem~atico que as t~ecnicas de desenvolvimento de programas. A fase de depura~ao ~e, em grande

parte dos casos, seguida sem método, fazendo tentativas cegas, tentando alguma coisa, qualquer coisa, pois não se sabe como deve ser abordada.

Os erros de um programa podem ser de dois tipos distintos, erros de natureza sintáctica e erros de natureza semântica.

A depuração sintáctica

Os erros de natureza sintáctica são os erros mais comuns em programação, e são os mais fáceis de localizar e de corrigir. Um erro sintáctico resulta da não conformidade de um constituinte do programa com as regras sintácticas da linguagem de programação. Os erros sintácticos podem ser causados por erros de ortografia ao escrevermos o programa ou por um lapso na representação da estrutura de uma instrução.

Os erros de natureza sintáctica são detectados pelo processador da linguagem, o qual produz mensagens de erro, indicando qual a instrução mais provável em que o erro se encontra e, normalmente, qual o tipo de erro verificado. A tendência actual em linguagens de programação é produzir mensagens de erro que auxiliem, tanto quanto possível, o programador.

A correcção dos erros sintácticos normalmente não origina grandes modificações no programa.

A fase de depuração sintáctica termina quando o processador da linguagem não

Ver a nota de rodapé na página 26.

CAPÍTULO 12. O DESENVOLVIMENTO DE PROGRAMAS

encontra quaisquer erros de natureza sintáctica no programa. O programador inexperiente tende a ficar eufórico quando isto acontece, não tendo a noção de que a verdadeira fase de depuração vai então começar.

A depuração semântica

Os erros semânticos resultam do facto de o programa, sintacticamente correcto, ter um significado para o computador que é diferente do significado que o programador desejava que ele tivesse. Os erros de natureza semântica podem causar a interrupção da execução do programa, ciclos infinitos de execução, a produção de resultados

errados, *etc.*

Quando, durante a fase de depuração semântica, se descobre a existência de um erro cuja localização ou origem não é facilmente detectável, o programador deverá recorrer metodicamente às seguintes técnicas de depuração (ou a uma combinação delas):

1. Utilização de programas destinados à depuração. Certos processadores de linguagens fornecem programas especiais que permitem fazer o rastreio⁸

automático do programa, inspeccionar o estado do programa quando o erro se verificou, alterar valores de variáveis, modificar instruções, *etc.*

2. Utilização da técnica da depuração da base para o topo⁹. Utilizando esta técnica, testam-se, em primeiro lugar, os módulos (por módulo entenda-se uma função correspondente a um subproblema) que estão ao nível mais baixo, isto é, que não são decomponíveis em subproblemas, e só quando um nível está completamente testado se passa ao nível imediatamente acima.

Assim, quando se aborda a depuração de um nível tem-se a garantia de que não há erros em nenhum dos níveis que ele utiliza e, portanto, que os erros que surgirem dependem apenas das instruções nesse nível.

Depois de detectado um erro de natureza semântica, terá de se proceder à sua correção. A correção de um erro de natureza semântica poderá variar desde casos extremamente simples a casos que podem levar a uma revisão completa ⁸Do dicionário da Porto Editora: rastear v.t. seguir o rasto de; rastreio s.m. acto de rastear; rasto s.m. vestígio que alguém, algum animal ou alguma coisa deixou no solo ou no ar, quando passou.

⁹Do inglês, “bottom-up debugging”.

12.3. A PROGRAMAC

~
,

AO DA SOLUC

~
,

AO

da fase de desenvolvimento da solução e, consequentemente, à criação de um novo programa.

12.3.2

A finalização

da documentação

ao

O desenvolvimento da documentação do programa deve começar simultaneamente com a formulação do problema (fase 1) e continuar à medida que se desenvolve a solução (fase 2) e se escreve o programa (fase 3). Nesta secção, vamos descrever o conteúdo da documentação que deve estar associada a um programa no instante em que ele é entregue à equipa de testes (ver Secção 12.4).

A documentação de um programa é de dois tipos: a documentação destinada aos utilizadores do programa, chamada documentação de utilização, e a documentação destinada às pessoas que irão fazer a manutenção do programa, a documentação técnica.

A documentação de utilização

A documentação de utilização tem a finalidade de fornecer ao utilizador a informação necessária para a correcta utilização do programa. Esta documentação inclui normalmente o seguinte:

1. Uma descrição do que o programa faz. Nesta descrição deve estar incluída a área geral de aplicação do programa e uma descrição precisa do seu comportamento. Esta descrição deve ser bastante semelhante à descrição desenvolvida durante a fase de análise do problema.
2. Uma descrição do processo de utilização do programa. Deve ser explicado claramente ao utilizador do programa o que ele deve fazer, de modo a poder utilizar o programa.
3. Uma descrição da informação necessária ao bom funcionamento do programa. Uma vez o programa em execução, vai ser necessário fornecer-lhe certa informação para ser manipulada. A forma dessa informação é descrita nesta parte da documentação. Esta descrição pode incluir a forma em que os ficheiros com dados devem ser

fornecidos ao computador (se for caso disso), o tipo de comandos que o programa espera receber durante o seu funcionamento, *etc.*

314

CAPÍTULO 12. O DESENVOLVIMENTO DE PROGRAMAS

4. Uma descrição da informação produzida pelo programa, incluindo por vezes a explicação de mensagens de erro.

5. Uma descrição, em termos não técnicos, das limitações do programa.

A documentação técnica

A documentação técnica fornece ao programador que irá modificar o programa a informação necessária para a compreensão do programa. A documentação técnica é constituída por duas partes, a documentação externa e a documentação interna.

A parte da documentação técnica que constitui a documentação externa descreve o algoritmo desenvolvido na fase 2 (desenvolvimento da solução), a estrutura do programa, as principais funções que o constituem e a interligação entre elas.

Deve ainda descrever os tipos de informação utilizados no algoritmo e a justificação para a escolha de tais tipos.

A documentação interna é constituída pelos comentários do programa. Os comentários são linhas ou anotações que se inserem num programa e que descrevem, em língua natural, o significado de cada uma das partes do programa.

Cada linguagem de programação tem a sua notação própria para a criação de comentários; por exemplo, em Python um comentário é tudo o que aparece numa linha após o carácter `#`. Para exemplificar a utilização de comentários em Python, apresenta-se de seguida a função factorial, juntamente com um comentário:

```
# Calcula o factorial de um número.
```

```
# N~
```

```
ao testa se o número é negativo
```

```
def factorial (n):
```

```
fact = 1
```

```
for i in range(n, 0, -1):
```

```
fact = fact * i
```

```
return fact
```

Os comentários podem auxiliar tremendamente a compreensão de um programa, e por isso a sua colocação deve ser criteriosamente estudada. Os comentários

12.4. A FASE DE TESTES

315

devem identificar secções do programa e devem explicar claramente o objectivo dessas secções e o funcionamento do algoritmo respectivo. É importante não sobrecarregar o programa com comentários, pois isso dificulta a sua leitura.

Certos programadores inexperientes tendem por vezes a colocar um comentário antes de cada instrução, explicando o que é que ela faz. Os comentários devem ser escritos no programa, à medida que o programa vai sendo desenvolvido, e devem reflectir os pensamentos do programador ao longo do desenvolvimento do programa. Comentários escritos depois de o programa terminado tendem a ser superficiais e inadequados.

Uma boa documentação é essencial para a utilização e a manutenção de um programa. Sem a documentação para o utilizador, um programa, por excepcional que seja, não tem utilidade, pois ninguém o sabe usar. Por outro lado, uma boa documentação técnica é fundamental para a manutenção de um programa.

Podemos decidir modificar as suas características ou desejar corrigir um erro que é descoberto muito depois do desenvolvimento do programa ter terminado.

Para grandes programas, estas modificações são virtualmente impossíveis sem uma boa documentação técnica.

Finalmente, uma boa documentação pode ter fins didácticos. Ao tentarmos desenvolver um programa para uma dada aplicação, podemos aprender bastante ao estudarmos a documentação de um programa semelhante.

A fase de testes

Depois de o processo de depuração semântica aparentemente terminado, isto é, depois de o programa ser executado e produzir resultados correctos, poderá ser posto em causa se o programa resolve o problema para que foi proposto para todos os valores possíveis dos dados. Para garantir a resposta afirmativa a esta questão, o programa é entregue a uma equipa de testes, a qual deverá voltar a verificar os testes de cada um dos módulos executados na fase anterior e, simultaneamente, verificar se todos os módulos em conjunto correspondem à solução acordada com o cliente. A equipa de testes deverá criar uma série de casos de teste para o sistema global (tal como foi descrito no documento de concepção global) e testar o bom funcionamento do programa, para todos estes casos.

316

CAPÍTULO 12. O DESENVOLVIMENTO DE PROGRAMAS

Os casos de teste deverão ser escolhidos criteriosamente, de modo a testarem todos os caminhos, ou rastos, possíveis através do algoritmo. Por exemplo, suponhamos que a seguinte função recebe três números correspondentes aos comprimentos dos lados de um triângulo e decide se o triângulo é equilátero, isósceles ou escaleno:

```
def classifica (l1, l2, l3):
```

```
    if l1 == l2 == l3:
```

```
        return 'Equilátero'
```

```
    elif (l1 == l2) or (l1 == l3) or (l2 == l3): return 'Isósceles'
```

```
    else:
```

```
        return 'Escaleno'
```

Embora esta função esteja conceptualmente correcta, existem muitas situações que esta não verifica, nomeadamente, se l_1 , l_2 e l_3 correspondem aos lados de um triângulo. Para isso ter-se-á de verificar as seguintes condições: 1. As variáveis l_1 , l_2 e l_3 têm valores numéricos.

2. Nenhuma das variáveis l_1 , l_2 e l_3 é negativa.

3. Nenhuma das variáveis l1, l2 e l3 é nula.

4. A soma de quaisquer duas destas variáveis é maior do que a terceira (num triângulo, qualquer lado é menor do que a soma dos outros dois).

Para além destas condições, os casos de teste deverão incluir valores que testam triângulos isósceles, equiláteros e escalenos.

Devemos notar que, para programas complexos, é impossível testar completamente o programa para todas as combinações de dados e portanto, embora estes programas sejam testados de um modo sistemático e criterioso, existe sempre a possibilidade da existência de erros não detectados pelo programador. Como disse Edsger Dijkstra (1930–2002), uma das figuras mais influentes do Século XX no que respeita a programação estruturada, o processo de testar um programa pode ser utilizado para mostrar a presença de erros, mas nunca para mostrar a sua ausência! (“Program testing can be used to show the presence of bugs, but never to show their absence!” [Dahl et al., 1972], página 6).

12.5. A MANUTENÇÃO

~
s

AO

317

Necessidades

Testes de

+

*

do cliente

aceitação

ao

*

✱

Documento

Testes de

+

*

de requisitos

sistema

*

✱

Documento

Testes de

+

*

de concepç~

ao

integraç~

ao

*

✱

Código

Testes de

+

✱ módulo

Figura 12.3: Níveis de testes.

Em resumo, os testes de um programa s~ao efectuados a vários n´iveis (Figura 12.3). Os testes de módulo s~ao efectuados pelo programador

durante a fase da programação da solução. Os testes de integração são efectuados pela equipa de testes, tendo em atenção o documento de concepção. Após a execução, com sucesso, dos testes de integração, a equipa de testes deverá também verificar se o sistema está de acordo com o documento dos requisitos, efectuando testes de sistema. Finalmente, após a entrega, o cliente efectua testes de aceitação para verificar se o sistema está de acordo com as suas necessidades.

Existem métodos para demonstrar formalmente a correcção semântica de um programa, discutidos, por exemplo, em [Dijkstra, 1976], [Hoare, 1972] e [Wirth, 1973], mas a sua aplicabilidade ainda se limita a programas simples e pequenos.

12.5

A manutenção

ao

Esta fase decorre depois de o programa ter sido considerado terminado, e tem duas facetas distintas. Por um lado, consiste na verificação constante da possibilidade de alterações nas especificações do problema, e, no caso de alteração de especificações, na alteração correspondente do programa. Por outro lado, consiste na correcção dos eventuais erros descobertos durante o funcionamento

318

CAPÍTULO 12. O DESENVOLVIMENTO DE PROGRAMAS

Análise do problema

Análise de requisitos

Planeamento do projecto



Desenvolvimento da solução

ao

Documento de concepção

ao



Programaç~

ao da soluç~

ao

Programa

Documentaç~

ao de utilizador



Testes

Programa

Documentaç~

ao externa



Documentos

Manutenç~

ao

de alteraç~

ao

Figura 12.4: Modelo de cascata e documentos associados.

do programa. Em qualquer dos casos, uma alteraç~ao no programa obriga a uma correspondente alteraç~ao na documentaç~ao. Um

programa com erros de documentação ou com a documentação desactualizada pode ser pior do que um programa sem documentação, porque encoraja o programador a seguir falsas pistas.

Segundo Frederick P. Brooks, Jr.¹⁰, o custo de manutenção de um programa é superior a 40% do custo total do seu desenvolvimento. Este facto apela a um desenvolvimento cuidado do algoritmo e a uma boa documentação, aspectos que podem diminuir consideravelmente o custo de manutenção.

12.6

Notas finais

Neste capítulo apresentámos de um modo muito sumário as fases por que passa o desenvolvimento de um programa. Seguimos o modelo da cascata, segundo o qual o desenvolvimento de um programa passa por cinco fases sequenciais.

Discutimos que a actividade desenvolvida em cada uma destas fases pode levar à detecção de deficiências em qualquer das fases anteriores, que deve então ser repetida.

Esta constatação leva a uma reformulação do modelo apresentado, a qual se indica na Figura 12.4. Nesta figura mostramos que o trabalho desenvolvido em qualquer fase pode levar a um retrocesso para fases anteriores. Indicamos também os principais documentos produzidos em cada uma das fases. Na fase da manutenção podem verificar-se grandes alterações ao programa, o que pode

12.6. NOTAS FINAIS

319

levar a um retrocesso para qualquer uma das fases de desenvolvimento.

O assunto que abordámos neste capítulo insere-se no campo da Engenharia Informática a que se chama Engenharia da Programação. Informação adicional sobre este assunto poderá ser consultada em [Jalote, 1997] ou [Sommerville, 1996].

A Engenharia da Programação é uma das áreas da Engenharia Informática em que existe mais regulamentação sobre as metodologias a utilizar. Em [Moore, 1998] encontra-se uma perspectiva global das normas existentes nesta disciplina.

CAPÍTULO 12. O DESENVOLVIMENTO DE PROGRAMAS

Capítulo 13

Estruturas lineares

“The time has come,” the walrus said, “to talk of many things: Of shoes and ships - and sealing wax - of cabbages and kings”

Lewis Carroll, *Alice’s Adventures in Wonderland* Neste capítulo apresentamos duas estruturas de informação, as pilhas e as filas, que partilham com os tuplos, com as cadeias de caracteres e com as listas a propriedade dos seus elementos estarem organizados numa sequência. Estas estruturas de informação são conhecidas por estruturas lineares.

13.1

Pilhas

As pilhas¹ são estruturas de informação constituídas por uma sequência de elementos. Os elementos da pilha são retirados pela ordem inversa pela qual foram colocados — os elementos são adicionados ao topo da pilha e são também retirados do topo da pilha (as pilhas correspondem a um comportamento que vulgarmente é designado por LIFO²). Numa pilha não podemos aceder nem inspeccionar nenhum elemento, para além do elemento no topo da pilha. As pilhas correspondem a pilhas de objectos físicos, por exemplo uma pilha de tabuleiros numa cantina ou uma pilha de livros.

¹Em inglês “stack”.

²Do inglês “Last In First Out”.

321

322

CAPÍTULO 13. ESTRUTURAS LINEARES

O conceito de pilha é muito comum em informática. Por exemplo, a sequência de ambientes locais criados durante a chamada a funções utiliza o conceito de pilha.

Uma pilha tem apenas um elemento que se distingue dos outros: o

elemento do topo da pilha. Apenas o elemento do topo da pilha pode ser acedido ou removido. Nenhum elemento pode ser adicionado, removido ou inspeccionado, a não ser com o recurso a operações que colocam no topo da pilha, removem do topo da pilha e inspeccionam o elemento no topo da pilha.

13.1.1

Operações

operações básicas para pilhas

Vamos agora considerar as operações básicas para o tipo de informação pilha.

Para isso, iremos considerar cada um dos grupos de operações a definir para um novo tipo abstracto de informação:

1. Construtores. Os construtores são operações que constroem pilhas.

Os construtores para o tipo pilha devem incluir uma operação que gera pilhas a partir do nada, à qual chamaremos nova pilha.

Um outro construtor será uma operação que recebe um elemento e uma pilha, e que insere esse elemento na pilha. No caso das pilhas, o que interessa capturar neste construtor é o facto de apenas podermos inserir elementos no topo da pilha. Tendo isto em atenção, vamos definir um construtor chamado empurra, que recebe como argumentos uma pilha e um elemento a inserir na pilha e que produz a pilha resultante da inserção do elemento no topo da pilha.

2. Selectores. Os selectores são operações que seleccionam partes de pilhas.

De acordo com o que dissemos sobre as características das pilhas, deveremos ter um selector que indica o elemento no topo da pilha e um selector que retira o elemento no topo da pilha.

Deste modo, a operação topo recebe como argumento uma pilha e devolve o elemento no topo da pilha. Se a pilha for vazia, esta operação é indefinida.

A operação tira recebe como argumento uma pilha e devolve a pilha em que o elemento no topo da pilha é retirado. Se a pilha for vazia, esta operação é indefinida.

3. Reconhecedores. Os reconhecedores são operações que identificam tipos de pilhas.

A operação pilha recebe como argumento um elemento de um tipo qualquer e decide se este pertence ao tipo pilha.

A operação pilha vazia recebe como argumento uma pilha e decide se esta corresponde à pilha vazia (a pilha gerada por nova pilha).

4. Testes. Os testes são operações que relacionam pilhas entre si.

A operação pilhas iguais recebe como argumentos duas pilhas e decide se estas são ou não iguais.

Em resumo, o tipo pilha tem as seguintes operações básicas (estas operações referem o tipo elemento, que corresponde ao tipo dos elementos da pilha): 1. Construtores:

— nova pilha : $\{\}$ $\rightarrow$ pilha

nova pilha() tem como valor uma pilha sem elementos.

— empurra : pilha $\times$ elemento $\rightarrow$ pilha

empurra(pilha, elm) tem como valor a pilha que resulta de inserir o elemento elm no topo da pilha pilha.

2. Selectores:

— topo : pilha $\diamond \rightarrow$ elemento

topo(pilha) tem como valor o elemento que se encontra no topo da pilha pilha. Se a pilha não contiver elementos, o valor desta operação é indefinido.

— tira : pilha $\diamond \rightarrow$ pilha

tira(pilha) tem como valor a pilha que resulta de remover o elemento que se encontra no topo da pilha pilha. Se a pilha não contiver elementos, o valor desta operação é indefinido.

3. Reconhecedores:

— pilha : universal $\diamond \rightarrow$ lógico

$\text{pilha}(\text{arg})$ tem o valor verdadeiro, se arg é uma pilha, e tem o valor falso, em caso contrário.

324

CAPÍTULO 13. ESTRUTURAS LINEARES

3

5

7

= = =

Figura 13.1: Representação externa para pilhas.

— pilha vazia : pilha $\diamond \rightarrow$ lógico

$\text{pilha vazia}(\text{pilha})$ tem o valor verdadeiro, se pilha é a pilha vazia, e tem o valor falso, em caso contrário.

4. Testes:

— pilhas iguais : pilha $\times$ pilha $\rightarrow$ lógico

pilhas iguais(pilha

) tem o valor verdadeiro, se pilha é igual a 1, pilha 2

1

pilha , e tem o valor falso, em caso contrário.

2

Devemos ainda estabelecer uma representação externa para pilhas. Vamos convencionar que uma pilha será apresentada ao mundo exterior com a sequência dos elementos que a constituem, em linhas separadas, começando com o elemento no topo da pilha e terminando com a cadeia de caracteres “= = =”. Assim, a fila com os elementos 3 5 7 (em que 3 está no topo da pilha) será representada como se indica na Figura 13.1. Seguindo a nossa convenção, a pilha vazia será representada por “= = =”.

13.1.2

Axiomatizaç~

ao

Entre as operações básicas para pilhas devem verificar-se as seguintes relações: pilha(nova pilha()) = verdadeiro

pilha(empurra(p, e)) = verdadeiro

$\diamond$ verdadeiro se p não for vazia

pilha(tira(p)) =

$\perp$

em caso contrário

pilha vazia(nova pilha()) = verdadeiro

pilha vazia(empurra(p, e)) = falso

13.1. PILHAS

325

$\text{topo}(\text{empurra}(p, e)) = e$

$\text{tira}(\text{empurra}(p, e)) = p$

|

|

|

| pilha vazia(p1)

se pilha vazia(p2)

|

{ pilha vazia(p

pilhas iguais(p

2)

se pilha vazia(p1)

1, p2) = |

|

| $\text{topo}(p1) = \text{topo}(p2)$ se $\text{tira}(p1) = \text{tira}(p2)$

|

| falso

em caso contrário

13.1.3

Representaç~

ao de pilhas

Representaremos pilhas recorrendo a listas:

1. Uma pilha vazia é representada pela lista vazia.

2. Uma pilha não vazia é representada por uma lista em que o primeiro elemento corresponde ao elemento no topo da pilha, o segundo elemento ao elemento imediatamente a seguir ao elemento no topo da pilha e assim sucessivamente.

Na Figura 13.2 apresentamos a visualização gráfica da nossa representação para pilhas.

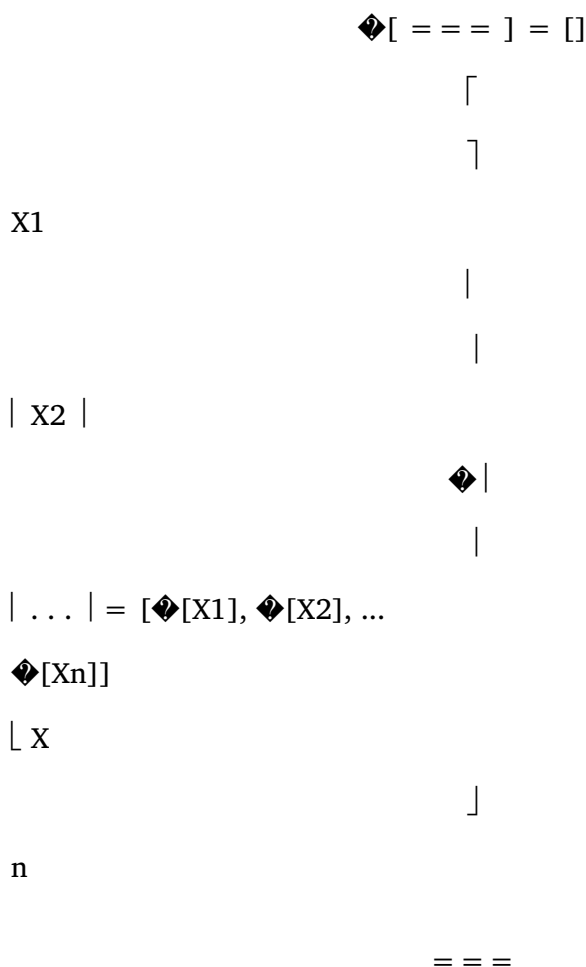


Figura 13.2: Representação de pilhas.

Realizaç~

ao das operaç~

oes básicas

Com base na representaç~ao escolhida para as pilhas, apresentamos a realizaç~ao das operaç~oes básicas para duas alternativas, a implementaç~ao de pilhas como funç~oes e a implementaç~ao de pilhas como objectos.

13.2.1

Pilhas recorrendo a funç~

oes

Ao apresentar a realizaç~ao de pilhas, vamos introduzir a instruç~ao try-except.

Esta instruç~ao tem a finalidade de “apanhar” os erros que s~ao detectados durante a execuç~ao de um grupo de instruç~oes (também conhecido como lidar com excepç~oes à execuç~ao de um grupo de instruç~oes), permitindo ao programador que o seu programa tome medidas excepcionais na presença destes erros.

A sintaxe da instruç~ao try-except é definida pela seguinte express~ao em notaç~ao BNF3:

◆instruç~ao try-except◆ ::= try: CR

◆instruç~ao composta◆

◆excepç~oes◆*

{finally: CR

◆instruç~ao composta◆}

◆excepç~oes◆ ::= except ◆excepç~ao◆: CR

◆instruç~ao composta◆

◆excepç~ao◆ ::= ◆nomes◆

O s~imbolo n~ao terminal ◆nomes◆ foi definido na página 74. A utilizaç~ao de nomes nesta instruç~ao diz respeito a nomes que

correspondem à identificação de vários tipos de erros que podem surgir num programa, alguns dos quais se mostram na Tabela 3.3, apresentada na página 90.

Ao encontrar a instrução:

try:

 ❖instrução❖

except❖excepção❖:

 ❖instrução❖

Esta não é uma definição completa da instrução try-except.

13.2. REALIZAÇÃO

~
s

AO DAS OPERAÇÕES

~
s

OES BÁSICAS

327

...

except❖excepção❖:

 ❖instrução❖

finally:

 ❖instrução❖

O Python começa por executar as instruções correspondentes a ❖instrução❖. Se estas instruções forem executadas sem gerar nenhum erro, a execução da instrução try-except está terminada. Se for gerado um erro de execução durante a execução das instruções correspondentes a ❖instrução❖, em vez de interromper a execução do programa, mostrando o erro ao utilizador, como é o comportamento normal do Python, este vai percorrer as exceções, começando por ❖excepção❖

... `except`, por esta ordem. Quando for encontrada a primeira `except` que corresponda ao erro gerado, digamos, `except`, o Python executa as instruções associadas a esta `except`, `finally`, terminando a execução da instrução `try-except`. Se nenhuma das `except` corresponder ao erro detectado, são executadas as instruções `finally`, terminando a execução da instrução `try-except`. Para exemplificar o funcionamento desta instrução, consideremos a interação:

```
> > > teste = [1, 2]
```

```
> > > teste[2] > 0
```

`IndexError: list index out of range`

```
> > > try:
```

```
...
```

```
teste[2] > 0
```

```
... except IndexError:
```

```
...
```

```
print('Enganou-se no indice')
```

```
...
```

Enganou-se no indice

```
> > >
```

Na realização das pilhas utilizamos a instrução `try-except` para detectar possí-

veis erros na utilização das pilhas. Por exemplo, se for solicitado o elemento no topo de uma pilha vazia, em lugar de deixar o Python gerar um erro correspondente a `IndexError`, o que iria de certo modo revelar a representação interna das pilhas, geramos uma mensagem apropriada.

Como alternativa, poderíamos ter escrito, por exemplo, a função como:

CAPÍTULO 13. ESTRUTURAS LINEARES

```
def topo (pilha):
```

```
    if pilha == []:
```

```
        raise ValueError ('A pilha n~
```

```
        ao tem elementos')
```

```
    else:
```

```
        return pilha[0]
```

Na nossa implementaç~ao de pilhas, iremos utilizar a instruç~ao try-except. As seguintes funç~oes criam o tipo pilha em Python recorrendo a funç~oes: def nova_pilha ():

```
    return []
```

```
def empurra (pilha, elemento):
```

```
    return [elemento] + pilha
```

```
def topo (pilha):
```

```
    try:
```

```
        return pilha[0]
```

```
    except IndexError:
```

```
        raise ValueError ('A pilha n~
```

```
        ao tem elementos')
```

```
def tira (pilha):
```

```
    try:
```

```
        return pilha[1:]
```

```
    except IndexError:
```

```
        raise ValueError ('A pilha n~
```

```
        ao tem elementos')
```

```
def pilha (x):
```

```
if x == []:
```

```
return True
```

```
elif isinstance(x, list):
```

```
return True
```

```
else:
```

```
return False
```

```
def pilha_vazia (pilha):
```

```
return pilha == []
```

13.2. REALIZAC

~
,

AO DAS OPERAC

~
,

OES B ÁSICAS

329

```
def mostra_pilha (pilha) :
```

```
if pilha != []:
```

```
for e in pilha:
```

```
print(e)
```

```
print(' = = =')
```

13.2.2

A classe pilha

Com base na representaç~ao da Figura 13.2, podemos alternativamente definir pilhas como objectos. Quando definimos as pilhas recorrendo a func~oes, as operaç~oes empurra e tira n~ao alteram a pilha que é seu argumento mas devolvem a pilha que resulta

da aplicação da operação. Definindo pilhas como objectos, estas operações modificam o estado interno do objecto que corresponde à pilha. Por esta razão, estas operações são conhecidas como modificadores. Decidimos que os modificadores não devolvem a pilha alterada, apenas efectuam um efeito, no entanto se for desejado que estas operações devolvam o valor da pilha deverá ser adicionada uma linha ao código com `return self.p`. A classe pilha está associada aos seguintes métodos:

```
class pilha:

# construtor

def __init__(self):

self.p = []

# modificadores

def empurra (self, elemento):

self.p = [elemento] + self.p

def tira (self):

try:

del(self.p[0])

except IndexError:

330
```

CAPÍTULO 13. ESTRUTURAS LINEARES

```
raise ValueError ('A pilha n~

ao tem elementos')

# selector

def topo (self):

try:

return self.p[0]

except IndexError:
```

raise ValueError ('A pilha n~

ao tem elementos')

reconhecedor

def pilha_vazia (self):

return self.p == []

representaç~

ao externa

def __repr__ (self):

if self.p != []:

rep = "

for e in self.p:

rep = rep + ' ' + str(e) + '\n'

rep = rep + '==='

return rep

else:

return '==='

Ao definir a classe pilha, propositadamente, n~ao definimos o método correspondente ao teste pilhas iguais. Este método iria comparar uma pilha que lhe é fornecida como argumento com a pilha que corresponde ao estado interno de uma instância do tipo pilha. Como este método n~ao tem acesso à representaç~ao interna da pilha que lhe é fornecida como argumento, a implementaç~ao deste teste exigia que a pilha fornecida como argumento fosse “desmanchada” para comparar os seus elementos com a pilha em que corresponde ao objecto em causa e depois fosse “construída” de novo, introduzindo um peso adicional na operaç~ao. Por esta raz~ao, decidimos n~ao definir este método.

13.3. EXPRESS ~

OES EM NOTAC

AO PÓS-FIXA

331

Com a classe pilha, podemos gerar a seguinte interação:

```
> > > p1 = pilha()
```

```
> > > p1
```

== =

```
> > > p1.empurra(3)
```

```
> > > p1.empurra(1)
```

```
> > > p1
```

1

3

== =

```
> > > p1.tira()
```

```
> > > p1
```

3

== =

13.3

Expressões

em notação

ao pós-fixa

Apresentamos uma das aplicações de pilhas, a avaliação de expressões em notação pós-fixa, uma notação em que o operador é escrito após os operandos. Por exemplo, a operação $2 + 8$ é escrita

em notação pós-fixa como $2\ 8\ +$. Expressões complexas são escritas, utilizando repetidamente este método. Por exemplo, $(4\ +\ 3\ \times\ 6)\ +\ 5\ \times\ 8$ é escrita

$$4\ 3\ 6\ \times\ +\ 5\ 8\ \times\ +$$

A notação pós-fixa apresenta a vantagem de não necessitar da especificação da precedência entre operadores e de não necessitar da utilização de parêntesis.

Para avaliar uma expressão em notação pós-fixa, percorremos a expressão da esquerda para a direita. Ao encontrar um operador, aplicamos esse operador aos dois últimos operandos percorridos e colocamos o seu valor na expressão (substituindo os operandos e o operador). Este processo é repetido, até que a expressão seja reduzida a um único valor, o qual representa o valor da expressão. Por exemplo, para calcular o valor da expressão em notação pós-fixa $4\ 3\ 6\ \times\ +\ 5\ 8\ \times\ +$, teremos de efectuar os seguintes passos:

332

CAPÍTULO 13. ESTRUTURAS LINEARES

$$4\ 3\ 6\ \times\ +\ 5\ 8\ \times\ +$$

$$4\ 18\ +\ 5\ 8\ \times\ +$$

$$22\ 5\ 8\ \times\ +$$

$$22\ 40\ +$$

62

Este processo de avaliação pode ser traduzido através de um algoritmo muito simples que recorre a uma pilha, a qual contém os operandos que foram encontrados ao percorrer a expressão. Utilizando este algoritmo começamos com uma pilha vazia (no início do algoritmo não encontramos nenhum operando).

Percorremos a expressão da esquerda para a direita. Sempre que encontramos um operando, este é colocado na pilha. Sempre que encontramos um operador, este é aplicado ao elemento no topo da pilha (o segundo operando), o qual é retirado da pilha, e ao novo elemento do topo da pilha (o primeiro operando), o qual também é retirado da pilha, e o resultado é colocado na pilha. A pilha tem assim o papel de “memorizar” os operandos que foram encontrados.

Apresentamos na Figura 13.3 a sequência de passos necessários para avaliar a expressão

$4 \ 3 \ 6 \times + \ 5 \ 8 \times + ,$

recorrendo a este algoritmo, bem como a indicação da pilha gerada. A seta indica qual o componente da expressão que está a ser analisado.

A seguinte função efectua a avaliação de expressões em notação pós-fixa, usando a classe pilha:

```
from pilhas import *  
  
def avalia ():  
  
    exp = input('Escreva uma expressão  
ao\n-> ')  
  
    operandos = pilha()  
  
    i = 0  
  
    while i < len(exp):  
  
        simbolo = "
```

13.3. EXPRESSÕES

OES EM NOTAC

, ~

AO PÓS-FIXA

333

436

$$\times + 58 \times +$$

$$= = =$$

*

4

436

$$\times + 58 \times +$$

$$= = =$$

*

3

4 3 6

$\times + 58 \times +$

4

*

$= = =$

6

3

4 3 6

$\times + 58 \times +$

4

*

= = =

18

4 3 6

$\times + 58 \times +$

4

*

= = =

22

436

$$\times + 58 \times +$$

$$= = =$$

*

5

4 3 6

$\times + 58 \times +$

22

*

= = =

8

5

4 3 6

$\times + 58 \times +$

22

*

= = =

40

4 3 6

$\times + 58 \times +$

22

*

= = =

62

4 3 6

$\times + 58 \times +$

$= = =$

*

Figura 13.3: Sequência de passos necessários para se avaliar a expressão $4\ 3\ 6\ \times\ +\ 5\ 8\ \times\ +$.

334

CAPÍTULO 13. ESTRUTURAS LINEARES

```
while i < len(exp) and exp[i] != '':
```

```
# obtém o s'
```

```
simbolo
```

```
simbolo = simbolo + exp[i]
```

```
i = i + 1
```

```
while i < len(exp) and exp[i] == '':
```

```
# salta brancos
```

```
i = i + 1
```

```
if simbolo in ['+', '-', '*', '/']:
```

```
op2 = operandos.topo()
```

```
operandos.tira()
```

```
op1 = operandos.topo()
```

```
operandos.tira()
```

```
if simbolo == '+':
```

```
res = op1 + op2
```

```
elif simbolo == '-':
```

```
res = op1 - op2
```

```
elif simbolo == '*':
```

```
res = op1 * op2
```

```
else:
```

```
res = op1 / op2
```

```
operandos.empurra(res)
```

```
else:
```

```
operandos.empurra(eval(simbolo))
```

```
res = operandos.topo()
```

```
print(' = ', res)
```

Com esta função podemos gerar a seguinte interação:

```
> > > avalia()
```

Escreva uma expressão

ao

```
-> 2 3 *
```

=

6

```
> > > avalia()
```

Escreva uma expressão

ao

```
-> 2 3 4 * +
```

=

14

> > > avalia()

Escreva uma express~

ao

13.4. BALANCEAMENTO DE PAR ÊNTESIS

335

-> 4 3 6 * + 5 8 * +

=

62

13.4

Balanceamento de parêntesis

Ao longo do livro temos vindo a utilizar expressões que contêm parêntesis. Um dos aspectos para a correcção destas expressões corresponde ao facto dos seus parêntesis estarem balanceados, ou seja cada parênteses que é aberto tem um parênteses fechado correspondente e os pares de parêntesis estão correctamente encadeados. Por exemplo, as expressões $(2 + 3 * 5$ e $(2 + 3)) * 5$ estão sintacticamente incorrectas pois na primeira não existe parênteses a fechar "("

e na segunda não existe um abrir parênteses correspondente a ")". O conceito de balanceamento de parêntesis estende-se naturalmente à utilização de outros tipos de parêntesis, "[" e "]" e "{" e "}".

As pilhas são estruturas de informação adequadas a avaliar o balanceamento de parêntesis numa expressão: percorrendo uma expressão da esquerda para a direita, sempre que é encontrado um abrir parêntesis este é colocado na pilha, o que significa que a pilha contém a sequência dos parêntesis que foram abertos, do mais recente para o mais antigo; sempre que é encontrado um fechar parênteses, consulta-se o elemento no topo da pilha, se este corresponder a um abrir parênteses do mesmo tipo ("(", "[" ou "{), então foi encontrado um par de parêntesis balanceados e o elemento no topo da pilha é retirado, em caso contrário podemos afirmar que os parêntesis não estão balanceados.

O seguinte programa efectua a verificação do balanceamento de

parêntesis numa expressão. É importante notar que, neste programa, qualquer símbolo que não seja um parênteses é ignorado. Isto significa que o programa pode também ser utilizado para verificar o balanceamento de uma cadeia de caracteres apenas contendo parêntesis. Como segunda observação, notemos que se for detectado que os parêntesis não estão balanceados, o ciclo de verificação termina imediatamente, pois não interessa continuar a processar a expressão.

Apresentamos a função que testa o balanceamento de parêntesis utilizando a implementação de pilhas usando funções.

```
def balanceados():
```

```
    balanc = True
```

```
336
```

CAPÍTULO 13. ESTRUTURAS LINEARES

```
pars = nova_pilha()
```

```
exp = input('Escreva uma express~
```

```
ao\n--> ')
```

```
for c in exp:
```

```
    if c in ['(', '[', '{']:
```

```
        pars = empurra(pars, c)
```

```
    elif c in [')', ']', '}']:
```

```
        if not pilha_vazia(pars):
```

```
            outro = topo(pars)
```

```
            if (c == ')') and outro == '(' or \
```

```
                (c == ']') and outro == '[' or \
```

```
                (c == '}') and outro == '{'):
```

```
                pars = tira(pars)
```

```
            else:
```

```
balanc = False
```

```
break
```

```
else:
```

```
balanc = False
```

```
break
```

```
if balanc and pilha_vazia(pars):
```

```
print('parentesis correctos')
```

```
else:
```

```
print('parentesis n~
```

```
ao balanceados')
```

Com este programa, geramos a seguinte interacç~ao:

```
> > > balanceados()
```

```
Escreva uma express~
```

```
ao
```

```
2 * [(5 + 7) * a + b]
```

```
parentesis correctos
```

```
> > > balanceados()
```

```
Escreva uma express~
```

```
ao
```

```
3)
```

```
parentesis n~
```

```
ao balanceados
```

```
> > > balanceados()
```

```
Escreva uma express~
```

ao

({[[O]]}O)

parentesis correctos

13.5. FILAS

337

13.5

Filas

As filas est~ao omnipresentes no nosso dia-a-dia. Começamos o dia em filas de transito, normalmente deparamo-nos filas quando vamos almoçar, existem filas em caixas de supermercados, em bilheteiras e em muitas outras situaç~oes do nosso quotidiano. Uma fila pode corresponder a uma fila f~isica de entidades ou de pessoas ou pode corresponder a uma fila conceptual em que os elementos têm uma indicaç~ao da sua ordem na fila, por exemplo as filas geradas numa loja que possui uma máquina para atribuir números de ordem a clientes.

Em programaç~ao, as filas s~ao estruturas de informaç~ao constituídas por uma sequ~encia de elementos. Os elementos da fila s~ao retirados pela ordem em que foram colocados. Uma fila tem dois elementos que se distinguem dos restantes, o elemento do in~icio da fila, o qual pode ser retirado da fila, e o elemento do fim da fila, a seguir ao qual ser~ao adicionados novos elementos à fila. Apenas um elemento da fila pode ser acedido ou removido, o elemento “mais antigo” da fila. Nenhum elemento pode ser adicionado, removido ou inspeccionado a n~ao ser com o recurso a operaç~oes que colocam no fim da fila, removem do in~icio da fila e inspeccionam o elemento no in~icio da fila. As filas correspondem a um comportamento que vulgarmente é designado por FIFO5.

13.5.1

Operaç~

oes básicas para filas

Consideremos as operaç~oes básicas para o tipo de informaç~ao fila: 1. Construtores.

Os construtores para o tipo fila devem incluir uma operação que gera filas a partir do nada, à qual chamaremos nova fila.

Um outro construtor será uma operação que recebe um elemento e uma fila, e que insere esse elemento na fila. O que interessa capturar neste construtor é o facto de apenas podermos inserir elementos no fim da fila.

Tendo isto em atenção, definimos um construtor chamado coloca, que recebe como argumentos uma fila e um elemento a inserir na fila e que produz a fila resultante da inserção do elemento no fim da fila.

4Em inglês “queue”.

5Do inglês “First In First Out”.

338

CAPÍTULO 13. ESTRUTURAS LINEARES

2. Selectores.

De acordo com o que dissemos sobre as características das filas, devemos ter um selector que indica o elemento no início da fila, e um selector que retira o primeiro elemento da fila. Definimos também um selector, comprimento que devolve o número de elementos da fila.

Deste modo, a operação inicio recebe como argumento uma fila e devolve o primeiro elemento da fila. Se a fila for vazia, esta operação é indefinida.

A operação retira recebe como argumento uma fila, e devolve a fila em que o elemento no início da fila é retirado. Se a fila for vazia, esta operação é indefinida.

A operação comprimento recebe como argumento uma fila, e devolve o número de elementos da fila.

3. Reconhecedores.

A operação fila recebe como argumento um elemento de um tipo qualquer e decide se este pertence ou não ao tipo fila.

A operação fila vazia recebe como argumento uma fila e decide se esta corresponde à fila vazia (a fila gerada por nova fila).

4. Testes.

A operação `filas iguais` recebe como argumento duas filas e decide se estas são ou não iguais.

Em resumo, o tipo `fila` tem as seguintes operações básicas (estas operações referem o tipo `elemento`, que corresponde ao tipo dos elementos da fila):

1. Construtores:

— `nova_fila : {} → fila`

`nova_fila()` tem como valor uma fila sem elementos.

— `coloca : fila × elemento → fila`

`coloca(fila, elm)` tem como valor a fila que resulta de inserir o elemento `elm` no fim da fila `fila`.

2. Selectores:

13.5. FILAS

339

— inicio : fila $\diamond \rightarrow$ elemento

inicio(fila) tem como valor o elemento que se encontra no início da fila fila. Se a fila não contiver elementos, o valor desta operação é indefinido.

— retira : fila $\diamond \rightarrow$ fila

retira(fila) tem como valor a fila que resulta de remover o elemento que se encontra no início da fila fila. Se a fila não contiver elementos, o valor desta operação é indefinido.

— comprimento : fila $\diamond \rightarrow$ inteiro

comprimento(fila) tem como valor o número de elementos da fila.

3. Reconhecedores:

— fila : universal $\diamond \rightarrow$ lógico

fila(arg) tem o valor verdadeiro, se arg é uma fila, e tem o valor falso, em caso contrário.

— fila vazia : fila $\diamond \rightarrow$ lógico

fila vazia(fila) tem o valor verdadeiro, se fila é a fila vazia, e tem o valor falso, em caso contrário.

4. Testes:

— filas iguais : fila $\times$ fila $\rightarrow$ lógico

filas iguais(fila

) tem o valor verdadeiro, se fila é igual a fila , e 1, fila 2

1

2

tem o valor falso, em caso contrário.

Devemos ainda estabelecer uma representação externa para filas. Vamos convencionar que uma fila será apresentada ao mundo exterior com a sequência dos elementos que a constituem, separados por espaços em branco. A totalidade dos elementos da fila será apresentada dentro de “<”, indicando a posição de inserção e remoção de elementos. Assim, a fila com os elementos 3 5 7 (em que 3 está no início da fila) será representada por:

< 3 5 7 <

Seguindo a nossa convenção, a fila vazia será representada por < <.

340

CAPÍTULO 13. ESTRUTURAS LINEARES

13.5.2

Axiomatização

ao

Entre as operações básicas para filas devem verificar-se as seguintes relações: $\text{fila}(\text{nova fila}()) = \text{verdadeiro}$

$\text{fila}(\text{coloca}(f, e)) = \text{verdadeiro}$

$\rightarrow$ verdadeiro se f não for vazia

$\text{fila}(\text{retira}(f)) =$

em caso contrário

fila vazia(nova_fila()) = verdadeiro

fila vazia(coloca(f, e)) = falso

❖ inicio(f) se f não for vazia

inicio(coloca(f, e)) =

e

em caso contrário

❖ coloca(retira(f),e) se f não for vazia

retira(coloca(f, e)) =

nova_fila()

em caso contrário

comprimento(nova_fila()) = 0

comprimento(coloca(f, e)) = 1 + comprimento(f)

{

|

| fila vazia(f

|

1)

se fila vazia(f2)

|

|

|

} fila vazia(f2)

se fila vazia(f1)

filas iguais(f1, f2) =

inicio(f

|

1) = inicio(f2)

se filas iguais(retira(f1),

|

|

|

|

retira(f

|

2))

| falso

em caso contrário

13.5.3

Representaç~

ao de filas

Representaremos filas recorrendo a listas. A representaç~ao de uma fila é definida do seguinte modo:

1. $\diamond[< <] = []$

2. $\diamond[< X_i \dots X_f <] = [\diamond[X_i], \dots$

$\diamond[X_f]]$

13.5. FILAS

341

13.5.4

A classe fila

Apenas definimos filas recorrendo a objectos. O estado interno de uma fila é definido através da variável `self.f` a qual contém a representação da fila como descrita na Secção 13.5.3. Na classe `Fila` decidimos definir um método correspondente a um transformador, `Fila para lista`, que produz uma lista contendo todos os elementos da fila. Um transformador é uma operação básica de um tipo abstracto de informação que transforma o tipo noutro tipo. São exemplos de transformadores as funções embutidas `str` e `int`. Com a definição deste transformador, deixamos de ter os problemas discutidos na página 330 e, por essa razão, podemos definir o método `Fila para lista`.

A classe `Fila` é definida através dos seguintes métodos: `class Fila`:

```
# construtor
```

```
def __init__(self):
```

```
    self.f = []
```

```
# modificadores (note-se que estes métodos alteram a fila) def coloca  
(self, elemento):
```

```
    self.f = self.f + [elemento]
```

```
def retira (self):
```

```
    try:
```

```
        del(self.f[0])
```

```
    except IndexError:
```

```
        print('retira: a fila n~
```

```
ao tem elementos')
```

```
# selectores
```

```
def inicio (self):
```

```
    try:
```

```
        return self.f[0]
```

```
    except IndexError :
```

CAPÍTULO 13. ESTRUTURAS LINEARES

```
print('inicio: a fila n~  
ao tem elementos')  
  
def comprimento (self):  
    return len(self.f)  
  
# reconhecedor  
  
def fila_vazia (self):  
    return self.f == []  
  
# teste  
  
def filas_iguais (self, outra):  
    outra_lista = outra.fila_para_lista()  
    if len(self.f) != len(outra_lista):  
        return False  
    else:  
        for i in range(len(self.f)):  
            if self.f[i] != outra_lista[i]:  
                return False  
        return True  
  
# transformador para listas  
  
def fila_para_lista (self):  
    lst = []  
    for i in range(len(self.f)):  
        lst = lst + [self.f[i]]
```

```
return lst
```

```
# representaç~
```

```
ao externa
```

```
def _repr_(self):
```

```
f = '< '
```

```
if self.f != []:
```

```
for i in range(len(self.f)):
```

```
f = f + self.f[i]._repr_() + ''
```

13.6. SIMULAC

~
,

AO DE UM SUPERMERCADO

343

```
f = f + '<'
```

```
return f
```

Com esta classe podemos gerar a seguinte interaç~ao:

```
>>> f = fila()
```

```
>>> f.retira()
```

retira: a fila n~

ao tem elementos

```
>>> f.inicio()
```

inicio: a fila n~

ao tem elementos

```
>>> f.coloca(1)
```

```
>>> f.coloca(2)
```

```
> > > f.coloca(3)
```

```
> > > f
```

```
< 1 2 3 <
```

```
> > > f.comprimento()
```

```
3
```

```
> > > f.retira()
```

```
> > > f
```

```
< 2 3 <
```

13.6

Simulaç~

ao de um supermercado

Uma das áreas de aplicaç~ao da informática, conhecida por simulaç~ao, consiste em “imitar ” a evoluç~ao do comportamento de um processo do mundo real. Para efectuar uma simulaç~ao é necessário construir um modelo do comportamento do processo, o qual captura as característias essenciais do processo e especifica o seu comportamento, estudando depois a evoluç~ao do processo ao longo do tempo. Um exemplo sofisticado de simulaç~ao corresponde a um simulador de voo, através do qual os pilotos s~ao treinados a lidar com situaç~oes adversas que podem ocorrer durante um voo.

Um programa de simulaç~ao tem pois como finalidade o estudo de uma situaç~ao hipotética do mundo real, avaliando o comportamento de um sistema ou de um processo em determinadas situaç~oes. Através da alteraç~ao de variáveis associadas à simulaç~ao, é possível prever o comportamento do sistema sob situaç~oes diversas.

344

CAPÍTULO 13. ESTRUTURAS LINEARES

Apresentamos um programa de simulaç~ao do tempo de espera nas caixas de um supermercado. Este programa corresponde a uma aplicaç~ao do tipo fila para representar a fila associada a cada caixa. As variáveis de simulaç~ao do nosso programa correspondem ao número de caixas abertas, a uma medida da afluência dos clientes ao

supermercado e a uma medida do número médio de compras previsto para os clientes. Alterando os valores destas variáveis é possível obter uma ideia do tempo que os clientes esperam para ser atendidos, eventualmente decidindo quantas caixas devem estar em funcionamento, em função da afluência e do número expectável de compras dos clientes.

O programa utiliza as seguintes variáveis para a simulação:

- **afluencia.** Caracteriza a afluência dos clientes ao supermercado. O seu valor corresponde a um inteiro entre 1 e 100, em que 1 representa a afluência mínima e 100 representa a afluência máxima.

- **apetencia.** Caracteriza a apetência dos clientes para fazerem compras.

O seu valor corresponde a um inteiro entre 1 e 100, em que 1 representa a apetência mínima e 100 representa a apetência máxima. Por exemplo, no final do mês assume-se que as pessoas fazem as compras do mês pelo que a sua apetência para fazer compras será maior do que no meio do mês. Quanto maior for a apetência para compras maior será o número de compras associadas a cada cliente.

- **n caixas.** Representa o número de caixas que estão abertas durante a simulação.

- **ciclos.** Esta variável define o tempo que demora a simulação. Este tempo é caracterizado por um inteiro. Em cada ciclo, poderá ou não ser inserido um cliente numa fila de uma caixa, dependendo do valor da afluência e são verificados quais os clientes nas filas das caixas cujas compras já foram todas processadas.

Antes de apresentar o programa que efectua a simulação, vamos discutir os tipos de informação que este utiliza:

- O tipo cliente corresponde a um cliente do supermercado. Cada cliente é caracterizado por um certo número de compras (self.items) e pelo instante (número do ciclo) em que foi colocado numa das filas de pagamento

13.6. SIMULAC

(este instante corresponde à variável `self.entrada`). A classe cliente é definida do seguinte modo:

```
class cliente:

    # construtor

    def __init__(self, items, entrada):

        self.items = items

        self.entrada = entrada

    # selectores

    def artigos(self):

        return self.items

    def tempo_entrada(self):

        return self.entrada

    # representaç~

ao externa

    def __repr__(self):

        return '[' + str(self.items) + \

        ']' + str(self.entrada) + ']'
```

- O tipo caixa corresponde a uma caixa aberta no supermercado. Este tipo é caracterizado por uma fila, tal como foi apresentada na Secção 13.5, a qual corresponde à fila dos clientes na caixa, e pela a seguinte informação adicional:

- O número da caixa;
- O instante a partir do qual a caixa está pronta para atender um cliente na fila. Se a caixa estiver a atender um cliente, esse instante corresponde ao instante em que o cliente deixa de ser atendido, se a fila da caixa não tem clientes, então esse instante é o instante actual;

– O número total de clientes que foi atendido pela caixa. Este valor

346

CAPÍTULO 13. ESTRUTURAS LINEARES

é aumentado em uma unidade sempre que a caixa termina o atendimento de um cliente;

– O número total de produtos processados pela caixa. Este número é contabilizado sempre que a caixa termina o atendimento de um cliente, somando o número de produtos que o cliente comprou ao total de produtos processados pela caixa;

– O tempo total de espera (medido em número de ciclos) associado a todos os clientes que foram atendidos pela caixa. Este número é contabilizado sempre que a caixa termina o atendimento de um cliente, somando o tempo de espera desse cliente ao tempo total de espera associado à caixa;

– O número de produtos que a caixa processa por unidade de tempo (entenda-se por ciclo). Este valor, entre 1 e 5, é calculado aleatoriamente sempre que uma caixa é aberta.

Em programação, é comum necessitar-se de gerar aleatoriamente números dentro de um certo intervalo. As sequências de números aleatórios produzidas por um programa não são aleatórias no verdadeiro sentido da palavra, na medida em que é possível prever qual será a sequência gerada, e por isso são denominadas pseudo-aleatórias.

Contudo, os números gerados podem ser considerados aleatórios porque não apresentam qualquer correlação entre si. Estas sequências de números aleatórios são repetidas sempre que o programa é executado de novo. Este facto não representa um inconveniente mas sim uma vantagem, uma vez que permite duplicar as condições de execução de um programa, o que é extremamente útil para a sua depuração.

Suponhamos então que pretendíamos gerar aleatoriamente números inteiros no intervalo $[1, n]$. A ideia básica na geração de números aleatórios consiste em gerar números uniformemente distribuídos no intervalo $[0, 1[$. Multiplicando o número gerado por n , obtemos um número real no intervalo $[0, n[$, finalmente desprezando a parte decimal do número gerado e adicionando 1 ao resultado, obtém-se um número inteiro no intervalo $[1, n]$.

Em Python, a função `random()`, localizada na biblioteca `random`, gera números aleatórios no intervalo $[0, 1[$. No nosso programa de simulação utilizamos a função `random`.

13.6. SIMULAC

~
,

AO DE UM SUPERMERCADO

347

O tipo caixa é realizado através da classe `caixa`, a qual apresenta o seguinte comportamento para alguns dos seus métodos (o comportamento dos restantes métodos é trivial):

- muda `info caixa`, recebe dois argumentos, o número de uma caixa, `nb caixa`, e um inteiro, `inst`, e actualiza o instante de tempo (`inst`) em que a caixa `nb caixa` está disponível para atender um cliente;
- aumenta clientes atendidos, recebe como argumento o número de uma caixa, `nb caixa`, e actualiza em uma unidade o número de clientes atendidos por essa caixa;
- muda `info produtos`, recebe dois argumentos, o número de uma caixa, `nb caixa` e um inteiro, `nb prods`, e actualiza com o valor `nb prods` o número de produtos processados pela caixa `nb caixa`;
- muda `info t espera`, recebe como argumentos o número de uma caixa (`nb caixa`) e uma unidade de tempo (`t`) e actualiza com o valor `t` o tempo acumulado de espera dos clientes atendidos pela caixa `nb caixa`.

A classe `caixa` é definida do seguinte modo6: `from filas import *`

```
from random import *
```

```
class caixa:
```

```
# construtor
```

```
def __init__(self, nb_cx, numero):
```

```
self.fila_cl = fila()
```

```
self.pronta_em = 0
```

```

self.cts_atend = 0

self.produtos_processados = 0

self.total_espera = 0

self.n_c = numero

# número de produtos tratados por unidade de tempo
self.prods_ut = int(random() * 5) + 1

```

6Esta classe necessita tanto do tipo filas como da função random.

348

CAPÍTULO 13. ESTRUTURAS LINEARES

```

# modificadores

def muda_info_caixa (self, valor):

self.pronta_em = valor

def aumenta_clientes_atendidos (self):

# muda o valor

self.cts_atend = self.cts_atend + 1

def muda_info_produtos (self, produtos):

self.produtos_processados = \

self.produtos_processados + produtos

def muda_info_t_espera (self, t):

self.total_espera = self.total_espera + t

# selectores

def fila_caixa (self):

return self.fila_cl

def num_caixa (self):

return self.n_c

```

```

def info_caixa (self):

return self.pronta_em

def clientes_atendidos (self):

return self.cts_atend

def info_produtos (self):

return self.produtos_processados

def info_t_espera (self):

return self.total_espera

```

13.6. SIMULAC

~
,

AO DE UM SUPERMERCADO

349

```

def produtos_processados_ciclo (self):

return self.prods_ut

# representaç~

ao externa

def _repr_ (self):

rep = 'Caixa ' + str(self.n_c) + ' ( ' + \

str(self.pronta_em) + \

'): ' + self.fila_cl._repr_()

return rep

```

O programa de simulaç~ao do supermercado é constituído por dois ciclos principais:

1. No primeiro, um ciclo for que é executado tantas vezes quanto o número de ciclos a simular, s~ao tratados os clientes que se encontram na filas das caixas (funç~ao trata clientes) e decide-se aleatoriamente

se deve ser gerado um novo cliente, com um número de compras que também é decidido de forma aleatória, colocando-o, de novo de forma aleatória, numa das filas de caixa.

2. O segundo ciclo, um ciclo while que é executado enquanto existirem clientes nas filas, aparece depois de todos os ciclos especificados na simulação terem sido executados. Este ciclo serve para processar os clientes que en-tretanto foram colocados nas filas das caixas mas que não foram atendidos.

Podemos considerar que este segundo ciclo surge depois das portas do supermercado já terem sido fechadas (não entram mais clientes) e é executado enquanto existem clientes por atender.

```
from random import *

from filas import *

from caixa import *

from cliente import *

def simula_supermercado (afluencia, apetencia, n_caixas, ciclos):

350

CAPÍTULO 13. ESTRUTURAS LINEARES

caixas = []

for i in range(n_caixas):

caixas = caixas + [caixa(n_caixas, i)]

for i in range(1, ciclos + 1):

print(' = CICLO = ', i)

# processa os clientes nas caixas

trata_clientes(i, caixas)

# decide a criação

ao de novo cliente e número de compras

aleatorio = random()
```

limiar = af luencia/100

```
if aleatorio < limiar: # um novo cliente é criado num_compras =  
int(random() 3 apetencia) + 1
```

```
print('--> Criado cliente com',num_compras, 'artigos') c =  
cliente(num_compras, i)
```

```
# insere o cliente na fila de uma caixa
```

```
nb_cx = int(random() * n_caixas) # selecciona a caixa cx =  
caixas[nb_cx]
```

```
fila = cx.fila_caixa()
```

```
fila.coloca(c)
```

```
mostra_caixas(caixas)
```

```
# processa as filas depois do fecho de entradas i = ciclos + 1
```

```
print('Entradas fechadas')
```

```
while existem_caixas_com_fila(caixas):
```

```
print(' == CICLO == ', i)
```

```
# processa os clientes nas caixas
```

```
trata_clientes(i, caixas)
```

```
mostra_caixas(caixas)
```

```
i = i + 1
```

```
processa_resultados(caixas)
```

13.6. SIMULAC

~
,

AO DE UM SUPERMERCADO

351

A funç~ao trata clientes corresponde ao atendimento dos clientes que est~ao no in ício das filas das caixas num determinado instante de tempo. Esta funç~ao é definida do seguinte modo:

```

def trata_clientes (tempo, caixas):

for i in range(len(caixas)):

cx = caixas[i]

if not cx.fila_caixa().fila_vazia():

# se a fila associada à caixa n~

ao é vazia

# verifica se o atendimento ao cliente terminou cliente =
cx.fila_caixa().inicio()

nb_artigos = cliente.artigos()

t_atendimento = tempo - cx.info_caixa()

artigos_prcls = \

nb_artigos/cx.produtos_processados_ciclo()

if artigos_prcls < t_atendimento:

# o cliente sai da fila da caixa

t_entrada_fila = cliente.tempo_entrada()

t_espera = tempo - t_entrada_fila

print('--> Processado cliente com ', \

nb_artigos, 'artigos na caixa', cx.num_caixa(),\

'tempo de espera', t_espera)

cx.muda_info_caixa(tempo + 1)

cx.aumenta_clientes_atendidos()

cx.muda_info_produtos(nb_artigos)

cx.muda_info_t_espera(t_espera)

f = cx.fila_caixa()

f.retira() # o cliente sai da fila

```


else:

para as caixas vazias actualiza o tempo potencial para

atendimento

cx.muda_info_caixa(tempo)

A função processa resultados mostra as estatísticas da execução do pro-

352

CAPÍTULO 13. ESTRUTURAS LINEARES

grama.

```
def processa_resultados(caixas):
```

```
    for i in range(len(caixas)):
```

```
        cx = caixas[i]
```

```
        c_t = cx.clientes_atendidos()
```

```
        if c_t != 0:
```

```
            print('Caixa', cx.num_caixa(), '(atendimento ' + \
                  str(cx.produtos_processados_ciclo()) + \
```

```
                  ' produtos por ciclo):')
```

```
            print(str(c_t) + ' clientes atendidos, ' + \
```

```
                  'média produtos/cliente ' + \
```

```
                  str(cx.info_produtos()/c_t) + \
```

```
                  ',\ntempo médio de espera ' + \
```

```
                  str(cx.info_t_espera()/c_t))
```

```
        else:
```

```
            print('Caixa ' + str(cx.num_caixa()) + \
```

```
            ': n~
```

ao atendeu clientes’)

A função mostra_caixas mostra em forma de texto a informação associada às caixas:

```
def mostra_caixas (cxs):
```

```
    for cx in cxs:
```

```
        print(cx)
```

A função existem_caixas_com_fila tem o valor verdadeiro apenas se existir alguma caixa cuja fila de clientes não seja vazia: def existem_caixas_com_fila (caixas):

```
    for i in range(len(caixas)):
```

```
        if not caixas[i].fila_caixa().fila_vazia():
```

```
            return True
```

```
    return False
```

A seguinte interação mostra parte dos resultados produzidos durante uma simulação:

13.6. SIMULAC

~
,

AO DE UM SUPERMERCADO

353

```
> > > simula_supermercado(100, 5, 5, 10)
```

```
= = CICLO = =
```

```
1
```

```
--> Criado cliente com 15 artigos
```

```
Caixa 0 (1): < [15:1] <
```

```
Caixa 1 (1): < <
```

```
Caixa 2 (1): < <
```

Caixa 3 (1): < <

Caixa 4 (1): < <

= = CICLO = =

2

--> Criado cliente com 8 artigos

Caixa 0 (1): < [15:1] <

Caixa 1 (2): < <

Caixa 2 (2): < [8:2] <

Caixa 3 (2): < <

Caixa 4 (2): < <

...

= = CICLO = =

10

--> Processado cliente com

2 artigos na caixa 2 tempo de espera 5

--> Criado cliente com 4 artigos

Caixa 0 (10): < [4:4] [4:9] <

Caixa 1 (10): < [4:10] <

Caixa 2 (11): < [13:6] [8:7] <

Caixa 3 (10): < <

Caixa 4 (8): < [10:8] <

Entradas fechadas

= = CICLO = =

11

Caixa 0 (10): < [4:4] [4:9] <

Caixa 1 (10): < [4:10] <

Caixa 2 (11): < [13:6] [8:7] <

Caixa 3 (11): < <

Caixa 4 (8): < [10:8] <

...

= = CICLO = =

24

--> Processado cliente com

8 artigos na caixa 2 tempo de espera 17

Caixa 0 (24): < <

Caixa 1 (24): < <

354

CAPÍTULO 13. ESTRUTURAS LINEARES

Caixa 2 (25): < <

Caixa 3 (24): < <

Caixa 4 (24): < <

Caixa 0 (atendimento 2 produtos por ciclo):

3 clientes atendidos, média produtos/cliente 7.666666666666667,
tempo médio de espera 8.333333333333334

Caixa 1 (atendimento 2 produtos por ciclo):

1 clientes atendidos, média produtos/cliente 4.0, tempo médio de
espera 3.0

Caixa 2 (atendimento 2 produtos por ciclo):

4 clientes atendidos, média produtos/cliente 7.75, tempo médio de
espera 9.75

Caixa 3 (atendimento 3 produtos por ciclo):

1 clientes atendidos, média produtos/cliente 2.0, tempo médio de espera 1.0

Caixa 4 (atendimento 2 produtos por ciclo):

1 clientes atendidos, média produtos/cliente 10.0, tempo médio de espera 6.0

13.7

Representaç~

ao gráfica

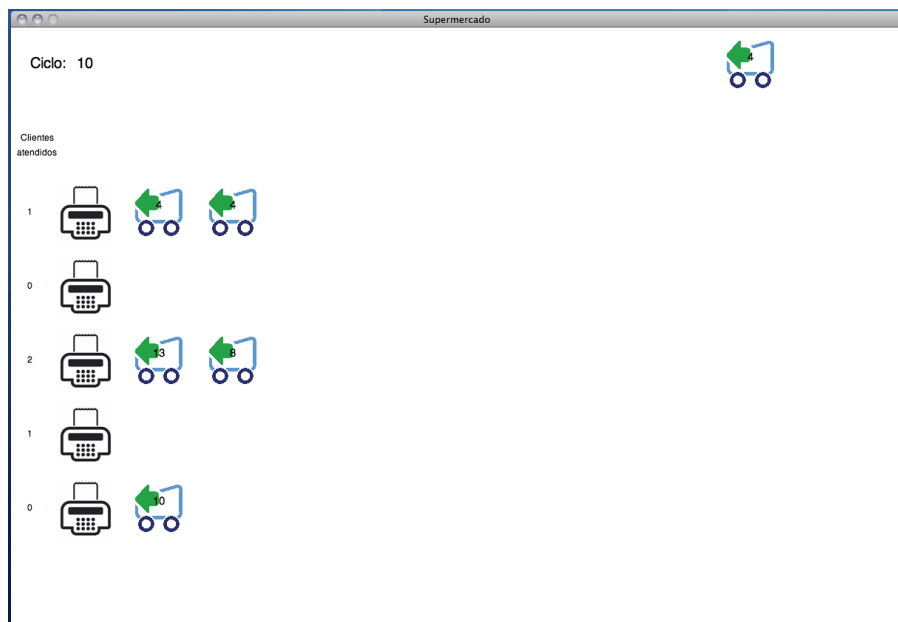
Em muitos programas de simulaç~ao é desejável apresentar de uma forma gráfica a evoluç~ao do processo simulado em vez de mostrar de forma textual o que está a acontecer, tal como o fizemos no exemplo anterior. Por exemplo, seria mais interessante que pudéssemos seguir a evoluç~ao da situaç~ao das filas das caixas através de uma janela como se mostra na Figura 13.4.

Nesta secç~ao apresentamos as modificaç~oes que é necessário introduzir no programa de simulaç~ao do supermercado de modo que este possa mostrar a evoluç~ao da simulaç~ao de uma forma gráfica. Para isso, utilizamos o módulo gráfico desenvolvido por John Zelle7.

Para a utilizaç~ao de uma janela gráfica, é necessário começar por criar um objecto que corresponde à janela a ser utilizada. A criaç~ao de uma janela é 7Este módulo pode ser obtido em <http://mcsp.wartburg.edu/zelle/python/graphics.py> e a sua documentaç~

ao está dispon ível em <http://mcsp.wartburg.edu/zelle/python/graphics/>

[graphics/index.html](http://mcsp.wartburg.edu/zelle/python/graphics/index.html).



13.7. REPRESENTAÇÃO GRÁFICA

~

AO GRÁFICA

355

Figura 13.4: Representação gráfica do supermercado no ciclo 9.

feita pela função `GraphWin`, cuja sintaxe é definida pelas seguintes expressões em notação BNF:

`GraphWin({título, largura, altura})`

`título` ::= cadeia de caracteres

`largura` ::= expressão

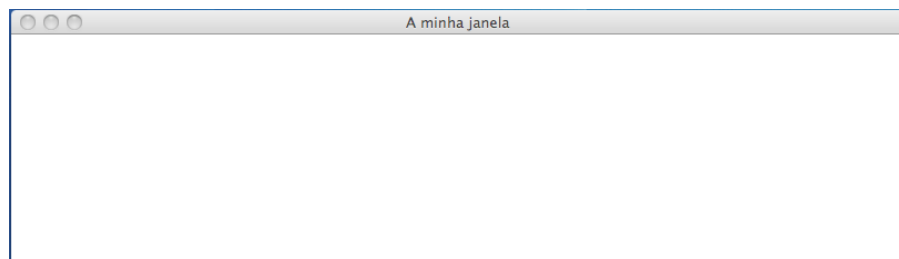
`altura` ::= expressão

Nesta função, `título` corresponde ao título que é colocado na janela e `largura`

e `altura` (expressões cujo valor é um inteiro positivo) especificam as dimensões da janela. Os parâmetros da função `GraphWin` são opcionais. Se não forem especificados, o Python assume os seguintes valores: 'Graphics Window', 200, 200. Por exemplo, a instrução `j =`

GraphWin('A minha janela', 800, 200) cria a janela apresentada na Figura 13.5, associando-a à variável `j`.

A partir do momento em que é criada uma janela correspondente a um objecto gráfico, é possível desenhar entidades nessa janela. Para isso convém saber que o conteúdo de uma janela (a zona onde podemos desenhar entidades) é constituído



356

CAPÍTULO 13. ESTRUTURAS LINEARES

Figura 13.5: Janela criada por GraphWin('A minha janela', 800, 200).

por um conjunto de pequenos pontos chamados pixeis 8. Na janela, os pixeis são referenciados através de um sistema de coordenadas cartesianas, cuja origem se situa no canto superior esquerdo da janela, aumentando a primeira coordenada da esquerda para a direita até ao valor máximo de $\text{largura} - 1$, e aumentando a segunda coordenada de cima para baixo até ao valor máximo de $\text{altura} - 1$. O

pixel localizado no ponto de coordenadas x e y é referenciado através da função `Point(x, y)`.

Associado a um objecto do tipo `GraphWin` existe um conjunto de métodos para manipular a janela, entre os quais:

- `Circle(ponto, raio)`, cria um objecto gráfico correspondente a um círculo, centrado no pixel correspondente ao `ponto` e com um raio (em número de pixeis) dado pela expressão `raio`. A criação deste objecto gráfico não origina o seu desenho na janela.
- `Rectangle(ponto1, ponto2)`, cria um objecto gráfico correspondente a um rectângulo, com vértices opostos localizados nos pixeis correspondentes aos pontos `ponto1` e `ponto2`. A criação deste objecto gráfico não origina o seu desenho na janela.

- `Text(ponto, texto)`, cria um objecto gráfico correspondente a texto, associado à cadeia de caracteres `texto`, o qual está centrado no pixel correspondente ao ponto `ponto`. A criação deste objecto gráfico não origina o seu desenho na janela.

- `Image(ponto, ficheiro)`, cria um objecto gráfico correspondente à imagem contida no ficheiro indicado pela cadeia de caracteres `ficheiro` (o qual deve estar em formato gif), centrada no pixel correspondente ao ponto.

9Note-se que a função `GraphWin` é o construtor da classe `GraphWin`.



13.7. REPRESENTAC

~
s

AO GR ÁFICA

357

(0, 0)

(200, 10)

(600, 100)

(100, 100)

400

(600, 300)

(300, 300)

(999, 399)

Figura 13.6: Objectos gráficos criados durante a interacção.

◆ponto◆. A criação deste objecto gráfico não origina o seu desenho na janela.

- `getMouse()`. Espera que o utilizador carregue com o rato na janela gráfica.
- `update()`. Força a execução das operações gráficas que possam estar pendentes.
- `close()`. Fecha a janela.

Cada objecto gráfico criado está, por sua vez, associado a métodos que controlam a sua manipulação na janela gráfica:

- `draw(◆janela◆)`, desenha o objecto gráfico correspondente na janela gráfica

◆janela◆.

- `undraw()`, apaga o objecto gráfico correspondente da janela gráfica onde este está desenhado.

Como exemplo, consideremos a seguinte interacção, na qual o ficheiro com o nome `clientepq.gif` corresponde à imagem apresentada no lado esquerdo da Figura 13.7:

```
> > > from graphics import *

> > > janela = GraphWin('A minha janela', 1000, 400)

> > > obj1 = Circle(Point(100, 100), 50)

358

CAPÍTULO 13. ESTRUTURAS LINEARES

> > > obj2 = Rectangle(Point(200, 10), Point(300, 300))

> > > obj1.draw(janela)

> > > obj2.draw(janela)

> > > icon = Image(Point(600, 300), 'clientepq.gif')
```

```
> > > icon.draw(janela)
```

```
> > > texto = Text(Point(600, 100), 'Cliente')
```

```
> > > texto.draw(janela)
```

Esta interacção origina os objectos gráficos apresentados na Figura 13.6, na qual também se mostram as coordenadas associadas a cada objecto.

Vamos agora discutir as alterações que é necessário introduzir no programa apresentado na Secção 13.6 de modo a que este apresente uma interface gráfica.

Estas envolvem uma mudança aos tipos de informação utilizados, de modo a que estes também contenham informação relativa aos objectos gráficos que os representam, e envolvem alterações ao próprio programa.

Começamos por considerar as alterações necessárias ao tipo cliente. Este tipo está definido no ficheiro com o nome clienteOG (cliente com objectos gráficos).

Recorde-se que um cliente é um objecto com um estado interno constituído por duas variáveis, representando o número de compras (self.items) e o instante de entrada na fila (self.entrada). Para utilizar a interface gráfica adicionamos mais uma variável ao seu estado interno, self.rep graf, correspondente à representação gráfica do cliente. Esta representação gráfica é constituída por dois objectos gráficos, uma representação do carrinho de compras, correspondente à imagem apresentada no lado esquerdo da Figura 13.710, e uma imagem gráfica da representação do número de compras do cliente. Assim, a variável self.rep graf está associada a um tuplo de dois elementos, contendo a representação de cada um destes objectos gráficos. Como a posição na janela em que aparece a representação de um cliente varia ao longo da execução do programa, decidimos não criar a representação gráfica no momento da criação do cliente, sendo esta apenas criada quando se sabe qual a posição na janela gráfica em que o cliente deve ser mostrado. Assim, existe um modificador associado à classe cliente, cria icons cliente, que cria os ícones associados a um cliente, altera o estado interno, e mostra esses ícones numa determinada posição da janela.

```
from graphics import *
```

10Esta imagem foi obtida de <http://findicons.com/search/check->

out/7.

13.7. REPRESENTAC

~
,

AO GR ÁFICA

359

class cliente:

constutores

def __init__(self, items, entrada, win):

self.items = items

self.entrada = entrada

def cria_icons_cliente (self, items, x, y, win):

objs_graf é um tuplo com dois elementos, o objecto

correspondente à representaç~

ao do carrinho e o

objecto correspondente à representaç~

ao das compras

ao criar a representaç~

ao gráfica, o objecto é

mostrado na janela

objs_graf = self.mostra_cliente(items, x, y, win) self.rep_graf =
(objs_graf[0], objs_graf[1]) return objs_graf

modificador

def apaga_cliente (self, icons, win):

remove da janela os icons correspondentes ao cliente
icons[0].undraw()

```
icons[1].undraw()
```

```
# selectores
```

```
def artigos (self):
```

```
return self.items
```

```
def tempo_entrada (self):
```

```
return self.entrada
```

```
def rep_grafica (self):
```

```
return self.rep_graf
```

```
360
```

CAPÍTULO 13. ESTRUTURAS LINEARES

```
# representaç~
```

```
ao externa
```

```
def __repr__ (self):
```

```
return '[' + str(self.items) + \
```

```
':' + str(self.entrada) + ']'
```

```
def mostra_cliente(self, n_compras, x, y, win): icon = Image(Point(x,  
y), 'clientepq.gif')
```

```
icon.draw(win)
```

```
texto = Text(Point(x, y-10), n_compras)
```

```
texto.setSize(15)
```

```
texto.draw(win)
```

```
return (icon, texto)
```

A classe caixa para o programa com interface gráfica existe no ficheiro com o nome caixasOG (caixas com objectos gráficos). Para além da informação que este tipo já continha (descrita na página 345), este tipo tem as seguintes variáveis de estado adicionais:

- Fila com os objectos gráficos existentes na fila da caixa. Estes objectos gráficos correspondem à representação de clientes que já foi apresentada;
- A posição, na janela gráfica, do primeiro cliente que se encontra na fila da caixa;
- A distância, na fila gráfica, entre os objectos correspondentes aos clientes;
- A posição na janela gráfica onde é mostrado o número de clientes atendidos pela caixa;
- O objecto gráfico correspondente ao número de clientes atendidos.

```
from filas import *
from random import *
from posicao import *
from graphics import *

class caixa:
```

13.7. REPRESENTAC

, ~

AO GR ÁFICA

361

construtor

```
def __init__(self, nb_cx, numero, win):
```

```
self.fila_cl = fila()
```

```
self.pronta_em = 0
```

```
self.cts_atend = 0
```

```
self.produtos_processados = 0
```

```
self.total_espera = 0
```

```
self.n_c = numero
```

```

# número de produtos tratados por unidade de tempo self.prods_ut =
int(random() * 5) + 1

self.f_graf = fila()

self.pos_p_cliente = ponto(200, \
750 - 100 * (nb_cx - numero))

self.d_c = 100

self.pos_nb_clientes = ponto(25, \
750 - 100 * (nb_cx - numero))

self.rep_c_a = Text(Point(self.pos_nb_clientes.pos_x(), \
self.pos_nb_clientes.pos_y()), \
0)

self.rep_c_a.draw(win)

# desenha icon correspondente à caixa

icon_caixa = \
Image(Point(100, 750 - 100 * (nb_cx - numero)), \
'caixapq.gif')

icon_caixa.draw(win)

# modificadores

def muda_info_caixa (self, valor):

self.pronta_em = valor

def aumenta_clientes_atendidos (self, win):

# muda o valor

362

CAPÍTULO 13. ESTRUTURAS LINEARES

self.cts_atend = self.cts_atend + 1

```

```

# apaga na interface gráfica valor antigo

self.rep_c_a.undraw()

# cria novo objecto e desenha-o

self.rep_c_a = Text(Point(self.pos_nb_clientes.pos_x(),\
self.pos_nb_clientes.pos_y()), \

self.cts_atend)

self.rep_c_a.draw(win)

win.update()

def muda_info_produtos (self, produtos):

self.produtos_processados = \

self.produtos_processados + produtos

def muda_info_t_espera (self, t):

self.total_espera = self.total_espera + t

def apaga_clientes (self, win):

while not self.f_graf.fila_vazia():

c = self.fila_cl.inicio()

rc = self.f_graf.inicio()

c.apaga_cliente(rc, win)

self.f_graf.retira()

# func~

oes de alto n´

ivel

def actualiza_interface (self, win):

# cria a fila gráfica representando os clientes na caixa

# obtém a lista com os clientes na fila

```

```
clientes = self.fila_cl.fila_para_lista()
```

```
for i in range(len(clientes)):
```

```
    cl = clientes[i]
```

```
    (carrinho, compras) = \
```

```
    cl.cria_icons_cliente(cl.artigos(),\
```

```
    self.pos_p_cliente.pos_x() \
```

```
13.7. REPRESENTAC
```

, ~

AO GR ÁFICA

363

```
+ i * self.d_c, \
```

```
self.pos_p_cliente.pos_y(), \
```

```
win)
```

```
win.update()
```

```
self.f_graf.coloca((carrinho, compras))
```

```
# selectores
```

```
def fila_caixa (self):
```

```
    return self.fila_cl
```

```
def num_caixa (self):
```

```
    return self.n_c
```

```
def info_caixa (self):
```

```
    return self.pronta_em
```

```
def clientes_atendidos (self):
```

```
    return self.cts_atend
```



```

def info_produtos (self):

return self.produtos_processados

def info_t_espera (self):

return self.total_espera

def produtos_processados_ciclo (self):

return self.prods_ut

def fila_graf_caixa (self):

return self.f_graf

def incremento_icons_caixas (self):

return self.d_c

def pos_novo_icon_fila (self):

x = self.pos_p_cliente.pos_x() + \

```



Figura 13.7: Ícones usados pelo programa.

```
self.f_graf.comprimento() * self.d_c

y = self.pos_p_cliente.pos_y()

return (x, y)

def __repr__(self):

rep = 'Caixa ' + str(self.n_c) + ' (' + \

str(self.pronta_em) + \

'): ' + self.fila_cl.__repr__()

return rep
```

Para além do ícone correspondente ao carrinho de compras (apresentado na parte esquerda da Figura 13.7), o programa usa um ícone para representar as caixas, o qual é apresentado na parte direita da Figura 13.711. Este ícone existe no ficheiro cujo nome é `caixapq.gif`.

O seguinte código define a classe ponto.

```
class ponto:

def __init__(self, x, y):

self.x = x

self.y = y

def pos_x(self):

return self.x

def pos_y(self):

return self.y

def __repr__(self):
```

11Obtido de <http://www.123rf.com/clipart-vector/cash-counter.html>.

AO GR ÁFICA

365

```
return '(' + str(self.x) + ', ' + str(self.y) + ')
```

O programa de simulação é alterado do seguinte modo: from random
import *

```
from filas import *
```

```
from caixaOG import *
```

```
from clienteOG import *
```

```
from graphics import *
```

```
def simula_supermercado (afluencia, apetencia, n_caixas, ciclos):
```

```
# define a janela para visualizaç~
```

```
ao dos resultados
```

```
win = GraphWin('Supermercado', 1200, 800)
```

```
mostra_janela_inicial(win, n_caixas, 0)
```

```
nbciclo = Text(Point(100, 50), 0)
```

```
nbciclo.setSize(20)
```

```
nbciclo.draw(win)
```

```
# desenha o cabeçalho dos clientes atendidos ca = Text(Point(35, 750  
- 100 * (n_caixas + 1)), \
```

```
'Clientes')
```

```
ca.draw(win)
```

```
ca2 = Text(Point(35, 750 - 100 * (n_caixas + 1) + 20), \
```

```
'atendidos')
```

```
ca2.draw(win)
```

```
# cria as filas vazias correspondentes às caixas e os
```

```
# respectivos objectos gráficos
```

```
scaixas = caixas(n_caixas, win)
```

```
win.update()
```

```
win.getMouse()
```

```
for i in range(1, ciclos + 1):
```

```
print(' == CICLO == ', i)
```

```
nbciclo = actualiza_ciclo (nbciclo, i, win)
```

```
# decide a criaç~
```

```
ao de novo cliente e número de compras
```

```
aleatorio = random()
```

```
366
```

CAPÍTULO 13. ESTRUTURAS LINEARES

```
limiar = afluencia/100
```

```
# processa os clientes nas caixas
```

```
trata_clientes(i, scaixas, win)
```

```
if aleatorio < limiar: # mm novo cliente é criado num_compras =  
int(random() * 3 + 1)
```

```
print('--> Criado cliente com ', num_compras, ' artigos') c =  
cliente(num_compras, i, win)
```

```
# mostra temporariamente o novo cliente na janela icons =  
c.cria_icons_cliente(num_compras, 1000, 50, win) win.getMouse()
```

```
# apaga o icon do novo cliente
```

```
c.apaga_cliente(icon, win)
```

```
# insere o cliente na fila de uma caixa
```

```
nb_cx = int(random() * n_caixas) # selecciona a caixa fila =
```

```
scaixas.fila_caixa(nb_cx)
```

```
fila.coloca(c)
```

```
fila_graf = scaixas.fila_graf_caixa(nb_cx)
```

```
(x, y) = scaixas.pos_novo_icon_fila(nb_cx, fila_graf) icons =  
c.cria_icons_cliente(num_compras, \
```

```
x, \
```

```
y, \
```

```
win)
```

```
fila_graf.coloca.icons)
```

```
print(scaixas)
```

```
# processa as filas depois do fecho de entradas i = ciclos + 1
```

```
cxsfechadas = Text(Point(400, 50), 'Portas Fechadas')  
cxsfechadas.setSize(20)
```

```
cxsfechadas.draw(win)
```

```
print('Entradas fechadas')
```

```
while scaixas.existem_caixas_com_fila():
```

```
print(' == CICLO == ', i)
```

13.7. REPRESENTAC

~
,

AO GR ÁFICA

367

```
nbciclo = actualiza_ciclo (nbciclo, i, win)
```

```
# processa os clientes nas caixas
```

```
trata_clientes(i, scaixas, win)
```

```
print(scaixas)
```

```
i = i + 1
```

```
processa_resultados(scaixas, n_caixas)
```

A função trata_clientes é alterada com a introdução de chamadas a funções que modificam a interface gráfica.

```
def trata_clientes (tempo, caixas, win):
```

```
for i in range(len(caixas)):
```

```
    cx = caixas[i]
```

```
    if not cx.fila_caixa().fila_vazia():
```

```
        # se a fila associada à caixa i não
```

```
        ao é vazia
```

```
        # verifica se o atendimento ao cliente terminou cliente =  
        cx.fila_caixa().inicio()
```

```
        nb_artigos = cliente.artigos()
```

```
        t_atendimento = tempo - cx.info_caixa()
```

```
        artigos_prcls = \
```

```
        nb_artigos/cx.produtos_processados_ciclo()
```

```
        if artigos_prcls < t_atendimento:
```

```
            # o cliente sai da fila da caixa
```

```
            t_entrada_fila = cliente.tempo_entrada()
```

```
            t_espera = tempo - t_entrada_fila
```

```
            print('--> Processado cliente com ', \
```

```
            nb_artigos, 'artigos na caixa', cx.num_caixa(), \
```

```
            'tempo de espera', t_espera)
```

```
            cx.muda_info_caixa(tempo + 1)
```

```
            cx.aumenta_clientes_atendidos(win)
```

```
cx.muda_info_produtos(nb_artigos)
```

```
cx.muda_info_t_espera(t_espera)
```

```
# actualiza a informaç~
```

```
ao gráfica
```

```
cx.apaga_clientes(win)
```

```
f = cx.fila_caixa()
```

```
368
```

CAPÍTULO 13. ESTRUTURAS LINEARES

```
f.retira() # o cliente sai da fila
```

```
cx.actualiza_interface(win)
```

```
else:
```

```
# para as caixas vazias actualiza o tempo potencial para
```

```
# atendimento
```

```
cx.muda_info_caixa(tempo)
```

A func~ao processa resultados n~ao é alterada, sendo igual à apresentada na página 351.

No programa com interface gráfica surgem as func~oes mostra janela inicial, que mostra na janela gráfica os ícones das caixas e o número de clientes atendidos, e actualiza ciclo, que actualiza na interface gráfica o número do ciclo do simulador.

```
def mostra_janela_inicial (win, n_caixas, ciclo): infociclos =  
Text(Point(50, 50), 'Ciclo:')
```

```
infociclos.setSize(20)
```

```
infociclos.draw(win)
```

```
win.update()
```

```
def actualiza_ciclo (nbciclo, i, win):
```

```
nbciclo.undraw()
```

```
nbciclo = Text(Point(100, 50), i)
```

```
nbciclo.setSize(20)
```

```
nbciclo.draw(win)
```

```
win.update()
```

```
win.getMouse()
```

```
return nbciclo
```

13.8

Notas finais

Apresentámos duas estruturas de informação cujos elementos apresentam uma ordem sequencial, as pilhas e as filas, bem como exemplos de aplicações que as utilizam.

Introduzimos o conceito de interface gráfica, utilizando um módulo gráfico muito

13.9. EXERCÍCIOS

369

simples. Existem vários módulos gráficos disponíveis para o Python, incluindo o wxPython (disponível em <http://www.wxpython.org/>) e o Tkinter (disponível a partir de <http://wiki.python.org/moin/TkInter>), os quais são mais potentes do que o módulo que apresentámos. No entanto, para a finalidade da nossa apresentação, o módulo gráfico que usámos apresenta a simplicidade e a funcionalidade necessárias.

13.9

Exercícios

13.1 Defina o tipo fila recorrendo a funções.

13.2 Uma fila de prioridades é uma estrutura de informação composta por um certo número de filas, cada uma das quais associada a uma determinada prioridade.

Suponha que desejava criar uma fila de prioridades com duas prioridades, urgente e normal. Nesta fila de prioridades, os novos

elementos s~ao adicionados à fila, indicando a sua prioridade, e s~ao colocados no fim da fila respectiva. Os elementos s~ao removidos da fila através da remoç~ao do elemento mais antigo da fila urgente. Se a fila urgente n~ao tiver elementos, a operaç~ao de remoç~ao remove o elemento mais antigo da fila normal. Existe uma operaç~ao para aumentar a prioridade, a qual remove o elemento mais antigo da fila normal e coloca-o como último elemento da fila urgente.

a) Especifique as operaç~oes básicas para o tipo fila de prioridades (com prioridades urgente e normal).

b) Escolha uma representaç~ao interna para o tipo fila de prioridades (com prioridades urgente e normal).

c) Com base na representaç~ao escolhida, escreva as operaç~oes básicas para o tipo fila de prioridades (com prioridades urgente e normal).

13.3 O programa de simulaç~ao do supermercado apresentado na Secç~ao 13.6

escolhe aleatoriamente a caixa em que vai ser inserido o cliente. Isto leva a situaç~oes em que clientes s~ao inseridos em caixas em que existe uma fila com clientes por atender num instante em que existem filas de caixa vazias. Modifique este programa de modo que um cliente é inserido na fila de caixa com menor número de clientes.

370

CAPÍTULO 13. ESTRUTURAS LINEARES

13.4 Em supermercados é vulgar a existência de caixas dedicadas ao atendimento de clientes com menos de 10 artigos. Modifique o programa da alínea anterior, de modo a que este contemple a existência deste tipo de caixas. Modifique também o seu algoritmo de inserç~ao de clientes em caixas tendo em atenç~ao este aspecto.

Capítulo 14

Árvores

So she went on, wondering more and more at every step, as everything turned into a tree the moment she came up to it.

Lewis Carroll, Through the Looking Glass

A árvore é um tipo estruturado de informação que é muito utilizado em programação. No dia-a-dia, utilizamos árvores para representar, por exemplo, a estrutura hierárquica de organizações e árvores genealógicas. Em informática, as árvores podem ser utilizadas para representar, entre muitas outras, estruturas de expressões (como foi feito nas figuras 1.3 a 1.6) e as funções invocadas durante a execução de um processo (como foi feito nas figuras 7.1 e 7.5).

Uma árvore¹ é um tipo que apresenta uma relação hierárquica entre os seus constituintes. A terminologia utilizada para designar os constituintes de uma árvore mistura termos provenientes das árvores que aparecem na natureza e termos provenientes de árvores genealógicas. Uma árvore pode ser vazia ou ser constituída por um elemento, a raiz da árvore, a qual domina, hierarquicamente, outras árvores. Uma árvore que apenas domina árvores vazias chama-se uma folha. As árvores dominadas chamam-se filhas da árvore dominadora, e esta árvore chama-se mãe das árvores dominadas. A ligação entre uma árvore e a árvore dominada é chamada um ramo da árvore.

Em programação, as árvores são normalmente escritas com a raiz no topo e com 1Em inglês “tree”.

371

372

CAPÍTULO 14. ÁRVORES

A

B

C

D

E

F

G

H

I

J

Figura 14.1: Exemplo de uma árvore.

as folhas na base. Na Figura 14.1, mostramos uma árvore cuja raiz é rotulada com A. Esta árvore domina três árvores: (1) uma árvore cuja raiz é rotulada com B e apenas domina árvores vazias; (2) uma árvore cuja raiz é rotulada com C; e (3) uma árvore cuja raiz é rotulada com D. A é a mãe de B, C e D; G é filha de D; as folhas da árvore são rotuladas com B, E, F, H, I e J.

Existe um caso particular de árvores, as árvores binárias, em que cada raiz domina exactamente duas árvores binárias. Uma árvore binária ou é vazia ou é constituída por uma raiz que domina duas árvores binárias, a árvore esquerda e a árvore direita. A árvore apresentada na Figura 14.1 não é binária, porque algumas das suas árvores (nomeadamente, as árvores cujas raízes são rotuladas com A e com G) dominam mais do que duas árvores. Na Figura 14.2, apresentamos uma árvore binária. É importante notar que a árvore cuja raiz é C domina à sua esquerda a árvore vazia e à sua direita a árvore cuja raiz é F.

Nesta secção, apenas consideramos árvores binárias. Isto não representa uma limitação séria, porque qualquer árvore pode ser transformada numa árvore binária². Daqui em diante, utilizaremos a palavra “árvore”, como sinónimo de árvore binária.

2Ver [Knuth, 1973a], Secção 2.3.2.

14.1. OPERAC

~
,

OES B ÁSICAS PARA ÁRVORES

373

A

B

C

E

D

F

H

G

Figura 14.2: Exemplo de uma árvore binária.

14.1

Operações

básicas para árvores

O primeiro passo para a criação do tipo árvore consiste em definir as suas operações básicas:

1. Construtores:

Os construtores para o tipo árvore devem incluir uma operação que gera árvores a partir do nada, à qual chamaremos nova arv.

O outro construtor é uma operação que recebe como argumentos uma raiz e duas árvores, e que cria uma árvore com essa raiz cujas árvores esquerda e direita são as árvores recebidas. Esta operação será chamada cria arv.

2. Selectores:

De acordo com o que dissemos sobre as características das árvores, deveremos ter um selector para escolher a raiz da árvore e selectores para a árvore esquerda e para a árvore direita. Estes selectores são chamados, respectivamente, raiz, arv esq e arv dir.

3. Reconhecedores:

A operação `arv` tem como argumento um elemento de qualquer tipo e decide se este corresponde ou não a uma árvore.

A operação `arv vazia` tem como argumento uma árvore e decide se esta corresponde ou não à árvore vazia (a árvore gerada por `nova arv`).

374

CAPÍTULO 14. ÁRVORES

4. Testes:

A operação `arv iguais` tem como argumentos duas árvores e decide se estas são iguais.

Em resumo, o tipo `árvore` tem as seguintes operações básicas, as quais se referem ao tipo `elemento` que corresponde ao tipo dos elementos da raiz: 1. Construtores:

- `nova arv : {} → árvore`

`nova arv()` tem como valor uma árvore vazia.

- `cria arv : elemento × árvore × árvore → árvore` cria `arv(raiz, aesq, adir)` tem como valor a árvore com raiz `raiz`, com árvore esquerda `aesq` e com árvore direita `adir`.

2. Selectores:

- $\text{raiz} : \text{árvore} \rightarrow \text{elemento}$

$\text{raiz}(\text{árv})$ recebe uma árvore, árv , e tem como valor a sua raiz. Se a árvore for vazia, o valor desta operação é indefinido.

- $\text{arv esq} : \text{árvore} \rightarrow \text{árvore}$

$\text{arv esq}(\text{árv})$ recebe uma árvore, árv , e tem como valor a sua árvore esquerda. Se a árvore for vazia, o valor desta operação é indefinido.

- $\text{arv dir} : \text{árvore} \rightarrow \text{árvore}$

$\text{arv dir}(\text{árv})$ recebe uma árvore, árv , e tem como valor a sua árvore direita. Se a árvore for vazia, o valor desta operação é indefinido.

3. Reconhecedores:

- arv : universal $\diamond \rightarrow$ lógico

arv(arg) tem o valor verdadeiro se arg é uma árvore e tem o valor falso, em caso contrário.

- arv vazia : árvore $\diamond \rightarrow$ lógico

arv vazia(árv) tem o valor verdadeiro se árv é uma árvore vazia e tem o valor falso, em caso contrário.

4. Testes:

14.1. OPERAC

~
,

OES B ÁSICAS PARA ÁRVORES

375

A

B

D

[]

[]

E

G

[]

[]

H

[]

[]

C

[]

F

[]

[]

Figura 14.3: Representa~ao externa da árvore da Figura 14.2.

• arv iguais : árvore $\times$ árvore $\rightarrow$ lógico

arv iguais(árv1, árv2) tem o valor verdadeiro se árv1 e árv2 são árvores iguais e tem o valor falso, em caso contrário.

De modo a utilizarmos o tipo árvore temos ainda de definir os transformadores de entrada e de saída, os quais transformam entre a representação que utilizamos para árvores, a representação externa, e a representação utilizada no nosso programa³. Seria interessante utilizar como representação externa para árvores uma representação gráfica como a que apresentamos nas figuras 14.1 e 14.2.

Para isso, poderemos utilizar as funções gráficas apresentadas na Secção 13.7.

Por uma questão de simplicidade, optamos por uma representação externa para árvores, tal como a que apresentamos na Figura 14.3, a qual corresponde à árvore da Figura 14.2. A nossa representação externa utiliza as seguintes convenções: (1) a árvore vazia é representada por []; (2) a árvore esquerda e a árvore direita de uma árvore são escritas em linhas diferentes (começando à mesma distância da margem esquerda), e dois espaços mais para a direita do que a sua raiz, que aparece na linha acima. A operação escreve arv recebe uma árvore e mostra-a de acordo com esta convenção.

³Novamente, não iremos definir o transformador de entrada.

376

CAPÍTULO 14. ÁRVORES

14.2

Axiomatização

ao

Entre as operações básicas devemos impor as seguintes condições (axiomatização), em que a, a1 e a2 são árvores e r é um elemento da raiz de uma árvore⁴:

arv(nova arv()) = verdadeiro

arv(cria arv(r, a1, a2)) = verdadeiro

arv vazia(nova arv()) = verdadeiro

$\text{arv vazia}(\text{cria arv}(r, a1, a2)) = f$ also

$\text{raiz}(\text{cria arv}(r, a1, a2)) = r$

$\text{arv esq}(\text{cria arv}(r, a1, a2)) = a1$

$\text{arv dir}(\text{cria arv}(r, a1, a2)) = a2$

$\text{cria arv}(\text{raiz}(a), \text{arv esq}(a), \text{arv dir}(a)) = a$ arv iguais(a, cria
 $\text{arv}(\text{raiz}(a), \text{arv esq}(a), \text{arv dir}(a))) = \text{verdadeiro}$ 14.3

Representaç~

ao de árvores

A escolha de uma representaç~ao interna para árvores vai ser influenciada pelo tipo de implementaç~ao que queremos utilizar para o tipo árvore. Podemos pensar em realizar o tipo árvore usando funções ou podemos realizar o tipo árvore como objectos.

14.3.1

Representaç~

ao para o uso de func~

oes

Ao implementar o tipo árvore através de funções, precisamos que as árvores estejam contidas numa única estrutura de informaç~ao, a qual será passada às várias funções que manipulam árvores. Uma das possíveis representaç~oes internas para árvores recorre a listas e poderá ser definida do seguinte modo (Figura 14.4): 1. Uma árvore vazia é representada lista vazia [].

4Por uma quest~ao de simplicidade, nas duas últimas equaç~oes omitimos a verificaç~ao de árvore n~

ao vazia.

14.3. REPRESENTAC

~
,

AO DE ÁRVORES

◆[Árvore vazia] = []

R



= [◆[R], ◆[A



d], ◆[Ae]]



Ae

Ad

Figura 14.4: Representação de árvores utilizando listas.

2. Uma árvore não vazia é representada por uma lista cujo primeiro elemento contém a raiz da árvore, cujo segundo elemento contém a representação da árvore esquerda e cujo terceiro elemento contém a representação da árvore direita.

Por exemplo, utilizando esta representação, a árvore da Figura 14.2, será representada pela lista [A, [B, [D, [], []], [E, [G, [], []], [H, [], []]]],

[C, [], [F, [], []]], ou, de uma forma mais intuitiva, baseando-nos no princípio que utilizamos para a representação externa:

[A,

[B,

[D, [], []],

[E,

[G, [], []],

[H, [], []]],

[C,

[],

[F, [], []]]]

14.3.2

Representaç~

ao para o uso com objectos

Sabemos que um objecto é uma entidade com estado interno.

Representado as árvores como objectos, o estado interno desses objectos deverá conter os diferentes constituintes de uma árvore. Neste caso, podemos utilizar três componentes

378

CAPÍTULO 14. ÁRVORES

$r :$

None

$\blacklozenge[\text{Árvore vazia}] = e : \perp$

$d :$

$\perp$

R

$r :$

$\blacklozenge[R]$

$\blacklozenge$

$\blacklozenge*$

$= e : \blacklozenge[A$

$\blacklozenge$

$e]$



d :

◆[A

A

d]

e

Ad

Figura 14.5: Representação de árvores recorrendo a objectos.

distintos para representar os três constituintes de uma árvore, a raiz, a árvore esquerda e a árvore direita. Embora estas três entidades sejam distintas, elas existem dentro das instâncias da classe árvore. Assim, uma árvore será representada por três variáveis, r, e e d, contendo, respectivamente, a raiz, a árvore esquerda e a árvore direita.

Para representar uma árvore vazia, vamos recorrer a um tipo de informação elementar existente em Python, o tipo None. O tipo None tem apenas uma constante, None, a qual representa “nada”. Já tínhamos apresentado este tipo, embora não explicitamente, quando na página 56 dissemos que a função print não devolve qualquer valor. A representação externa desta constante não é mostrada pelo Python quando uma expressão tem o valor None como o mostra a seguinte interacção:

```
> > > None
```

```
> > >
```

Assim, a representação que escolhemos para árvores é apresentada na Figura 14.5

(recorde-se que $\perp$ representa indefinido).

14.4

Realização

ao das operaç~

oes básicas

Apresentamos a realizaç~ao das operaç~oes básicas para as duas alternativas apresentadas na Secç~ao 14.3.

14.4. REALIZAC

~
,

AO DAS OPERAC

~
,

OES B ÁSICAS

379

14.4.1

Árvores recorrendo a funç~

oes

construtores

def nova_arv ():

return []

def cria_arv (r, a_e, a_d):

if arvore(a_e) and arvore(a_d):

return [r, a_e, a_d]

else:

raise ValueError ('cria_arv: o segundo e terceiro \ argumentos devem
ser árvores')

selectores

def raiz (a):

if a == []:

```
raise ValueError ('raiz: a árvore é vazia')
```

```
else:
```

```
return a[0]
```

```
def arv_esq(a):
```

```
if a == []:
```

```
raise ValueError ('arv_esq: a árvore é vazia') else:
```

```
return a[1]
```

```
def arv_dir (a):
```

```
if a == []:
```

```
raise ValueError ('arv_esq: a árvore é vazia') else:
```

```
return a[2]
```

```
# reconhecedores
```

```
def arvore (x):
```

```
380
```

CAPÍTULO 14. ÁRVORES

```
if isinstance(x, list):
```

```
if x == []:
```

```
return True
```

```
elif len(x) == 3 and \
```

```
arvore(x[1]) and \
```

```
arvore(x[2]):
```

```
return True
```

```
else:
```

```
return False
```

```

else:

return False

def arv_vazia (a):

return a == []

# testes

def arv_iguais (a1, a2):

if arv_vazia(a1):

return arv_vazia(a2)

elif raiz(a1) == raiz(a2):

return arv_iguais(arv_esq(a1), arv_esq(a2)) and \
arv_iguais(arv_dir(a1), arv_dir(a2))

else:

return False

# transformador de sa´

ida

def escreve (a):

def escreve_aux (a, indent):

if arv_vazia(a):

print(' ' * indent, '[]')

else:

print(' ' * indent, raiz(a))

escreve_aux(arv_esq(a), indent + 2)

escreve_aux(arv_dir(a), indent + 2)

```

14.4. REALIZAC

OES B ÁSICAS

381

escreve_aux(a, 0)

14.4.2

A classe árvore

Recordemos que definimos dois construtores para árvores, a operação nova arv o que gera árvores a partir do nada e a operação cria arv que recebe como argumentos uma raiz e duas árvores, e que cria uma árvore com essa raiz cujas árvores esquerda e direita são as árvores recebidas. Existem pois dois modos distintos de construir árvores, modos esses que devem ser contemplados pelo método init : se não lhe forem fornecidos argumentos, então init deve construir uma árvore vazia; se lhe forem fornecidos uma raiz e duas árvores, então init constrói uma árvore não vazia. Precisamos pois de definir uma função, init , que recebe um número variável de argumentos.

A função embutida print, que temos utilizado nos nossos programas, aceita qualquer número de argumentos. A execução de print() origina uma linha em branco, e a execução de print com qualquer número de argumentos origina a escrita dos valores dos seus argumentos seguidos de um salto de linha. Até agora, as funções que escrevemos têm um número fixo de argumentos, o que contrasta com algumas das funções embutidas do Python, por exemplo, a função print.

Antes de apresentar a classe arvore, vamos considerar um novo aspecto na definição de funções correspondente à utilização de um número arbitrário de elementos. O Python permite a definição de funções que aceitam um número arbitrário de argumentos. Para podermos utilizar este aspecto, teremos que rever a definição de função apresentada na página 74. Uma forma mais completa da definição de funções em Python é dada pelas seguintes expressões em notação BNF:

◆definição de função◆ ::= def ◆nome◆ (◆parâmetros formais◆)
: CR

TAB ◆corpo◆

❖parâmetros formais❖ ::= ❖nada❖ { *❖nome❖} ❖nomes❖{,
*❖nome❖}

A diferença desta definição em relação à definição apresentada na página 74, corresponde à possibilidade dos parâmetros formais terminarem com um nome que é antecedido por *. Por exemplo, esta definição autoriza-nos a criar uma função e Python cuja primeira linha corresponde a “def exfn(a, b, *c):”.

382

CAPÍTULO 14. ÁRVORES

Ao encontrar uma chamada a uma função cujo último parâmetro formal seja antecedido por um asterisco, o Python associa os $n - 1$ primeiros parâmetros formais com os $n - 1$ primeiros parâmetros concretos e associa o último parâmetro formal (o nome que é antecedido por um asterisco) com o tuplo constituído pelos restantes parâmetros concretos. A partir daqui, a execução da função segue os mesmos passos que anteriormente.

Consideremos a seguinte definição de função: def exfn (a, b, *c):

```
print('a = ', a)
```

```
print('b = ', b)
```

```
print('c = ', c)
```

Esta função permite gerar a seguinte interação, a qual revela este novo aspecto da definição de funções:

```
> > > exfn(3, 4)
```

```
a =
```

```
3
```

```
b =
```

```
4
```

```
c =
```

```
()
```

```
> > > exfn(5, 7, 9, 1, 2, 5)
```

```
a =
```

```
5
```

```
b =
```

```
7
```

```
c =
```

```
(9, 1, 2, 5)
```

```
> > > exfn(2)
```

TypeError: exfn() takes at least 2 arguments (1 given) A função exfn aceita, no mínimo, dois argumentos. Os dois primeiros parâmetros concretos são associados com os parâmetros formais a e b e tuplo contendo os restantes parâmetros concretos é associado ao parâmetro formal c.

Deste modo, o método init para a classe arvore deve permitir a utilização de zero ou três argumentos.

```
class arvore:
```

```
def __init__(self, *args):
```

14.4. REALIZAC

```
    ~  
    ,
```

AO DAS OPERAC

```
    ~  
    ,
```

OES B ÁSICAS

```
383
```

```
if args == ():
```

```
self.r = None
```

```
else:
```

```
if len(args) == 3:
```

```
if isinstance(args[1], arvore) and \
```

```
isinstance(args[2], arvore):

self.r = args[0]

self.e = args[1]

self.d = args[2]

else:

raise ValueError ('arvore: o segundo e \
terceiro argumentos devem ser arvores')

else:

raise ValueError ('arvore: aceita zero ou \
tr^
es argumentos')

def raiz (self):

if self.r == None:

raise ValueError ('raiz: a arvore é vazia')

else:

return self.r

def arv_esq (self):

if self.e == None:

raise ValueError ('arv_esq: a arvore é vazia') else:

return self.e

def arv_dir (self):

if self.d == None:

raise ValueError ('arv_dir: a arvore é vazia') else:

return self.d
```

```
def arv_vazia (self):
```

```
    return self.r == None
```

```
384
```

CAPÍTULO 14. ÁRVORES

```
def __repr__ (self):
```

```
    def __repr__aux (a, indent):
```

```
        if a.r == None:
```

```
            return ' ' * indent + '[' + '\n'
```

```
        else:
```

```
            return ' ' * indent + \
```

```
                '[' + str(a.r) + '\n' + \
```

```
                    __repr__aux(a.e, indent + 2) + \
```

```
                    __repr__aux(a.d, indent + 2) + \
```

```
                ' ' * (indent + 2) + ']' + '\n'
```

```
    return __repr__aux(self, 0)
```

```
14.5
```

Ordenaç~

ao por árvore

Nesta secç~ao, apresentamos uma aplicaç~ao que utiliza árvores, a ordenaç~ao por árvore. A ordenaç~ao por árvore é um algoritmo de ordenaç~ao muito eficiente, executado em dois passos sequenciais: primeiro coloca os elementos a serem ordenados numa árvore binária (chamada árvore binária de procura), depois percorre esta árvore seguindo um método determinado, “visitando” as raízes das árvores que a constituem. Isto é feito de modo a que a primeira raiz visitada contenha o menor elemento, a segunda raiz visitada, o elemento que é imediatamente maior, e assim sucessivamente.

A árvore binária é construída do seguinte modo. Começando com

uma árvore vazia, inserimos nesta árvore os elementos a serem ordenados, um de cada vez.

1. Um elemento é inserido numa árvore vazia através da criação de uma árvore cuja raiz contém o elemento a inserir e em que as árvores esquerda e direita são vazias.

2. Um elemento é inserido numa árvore não vazia comparando o elemento com a raiz da árvore. Se o elemento a ser inserido for maior do que o elemento que se encontra na raiz da árvore, o elemento é inserido, utilizando o mesmo método, na árvore direita da árvore inicial, caso contrário, é inserido, pelo mesmo método, na árvore esquerda.

14.5. ORDENAC

~
,

AO POR ÁRVORE

385

5

14

Figura 14.6: Árvore depois da inserção de 5 e de 14.

5

14

8

Figura 14.7: Árvore depois da inserção de 8.

Como exemplo desta função, suponhamos que desejávamos criar uma árvore binária de procura com os elementos 5, 14, 8, 2 e 20.

Em primeiro lugar, criamos uma árvore vazia, onde inserimos o elemento 5, criando uma árvore cuja raiz contém 5 e em que as árvores esquerda e direita são vazias. Seguidamente, inserimos o elemento 14 nesta árvore. Uma vez que $14 > 5$ (5 é a raiz da árvore), 14 é inserido na árvore da direita, originando a árvore apresentada na Figura 14.6. O elemento seguinte a ser inserido é 8.

Uma vez que $8 > 5$, este elemento será introduzido na árvore da

direita. Para inserir este elemento na árvore da direita, este é comparado com a raiz, e, uma vez que $8 < 14$, este é inserido na árvore da esquerda, dando origem à árvore apresentada na Figura 14.7. Em seguida, insere-se o número 2. Uma vez que $2 < 5$, este elemento será inserido na árvore da esquerda, dando origem à árvore apresentada na Figura 14.8. Finalmente, inserimos o elemento 20, dando origem à árvore apresentada na Figura 14.9.

Uma vez construída a árvore, teremos de a percorrer, visitando as raízes das suas árvores. As raízes devem ser visitadas por ordem crescente dos elementos existentes na árvore.

386

CAPÍTULO 14. ÁRVORES

5

2

14

8

Figura 14.8: Árvore depois da inserção de 2.

5

2

14

8

20

Figura 14.9: Árvore depois da inserção de 20.

O algoritmo para percorrer a árvore é o seguinte: 1. Percorrer uma árvore vazia não causa nenhuma ação.

2. Para percorrer uma árvore não vazia, primeiro percorremos a sua árvore da esquerda, depois visitamos a raiz, depois percorremos a sua árvore da direita.

Por exemplo, considerando a árvore da Figura 14.9, uma vez que esta árvore não é vazia, teremos primeiro de percorrer a sua árvore da

esquerda, depois visitar a raiz e depois percorrer a árvore da direita. Para percorrer a árvore da esquerda (cuja raiz contém 2), teremos de percorrer a árvore da esquerda (que é vazia), visitar a raiz, que contém o valor 2, e depois percorrer a árvore da direita, que também é vazia. Acabamos assim de percorrer a árvore cuja raiz contém 2, visitamos agora a raiz da árvore original, que contém 5, e percorremos a árvore da direita. O processo repete-se para a árvore da direita. Deixamos como

14.5. ORDENAC

~
,

AO POR ÁRVORE

387

exercício a verificação de que os elementos são visitados por ordem crescente do seu valor.

A função ordena_arvore recebe uma lista contendo valores a ordenar e devolve a lista correspondente à ordenação dos elementos recebidos. Esta função utiliza a definição de árvores como objectos.

```
from arvoresobj import *

def ordena_arvore (lst):

    return percorre(lista_para_arvore(lst))

def lista_para_arvore (lst):

def insere_arv (lst, arv):

def insere_elemento(el, arv):

    if arv.arv_vazia():

        return arvore(el, arvore(), arvore())

    elif el > arv.raiz():

        return arvore(arv.raiz(), \

            arv.arv_esq(), \

            insere_elemento(el, arv.arv_dir()))
```

```

else:

return arvore(arv.raiz(), \

insere_elemento(el, arv.arv_esq()), \

arv.arv_dir())

if lst == []:

return arv

else:

return insere_arv(lst[1:], \

insere_elemento(lst[0], arv))

return insere_arv(lst, arvore())

```

388

CAPÍTULO 14. ÁRVORES

```

def percorre (arv) :

if arv.arv_vazia() :

return []

else :

return percorre(arv.arv_esq()) + \

[arv.raiz()] + \

percorre(arv.arv_dir())

```

Com a qual podemos gerar a interacção:

```

>>> from arvoresobj import *

>>> ordena_arvore([5, 14, 8, 2, 20])

[2, 5, 8, 14, 20]

```

Apresentámos o tipo árvore que é muito utilizado em programação. Informação adicional sobre árvores e algoritmos que as manipulam podem ser consultados em [Knuth, 1973b] e [Cormen et al., 2009].

14.7

Exercícios

14.1 Defina a representação externa para árvores utilizando uma interface gráfica.

Capítulo 15

Epílogo

‘Would you tell me, please, which way I ought to go from here?’

‘That depends a good deal on where you want to get to,’ said the Cat.

Lewis Carroll, *Alice’s Adventures in Wonderland* No início dos anos 80, com o advento do computador pessoal, a utilização dos computadores banalizou-se. Num artigo escrito para o grande público, a revista TIME, em 1984, qualificou os programas (o chamado “software”) como o feiticeiro dentro da máquina. O artigo era iniciado do seguinte modo: “Hardware ou software?, pode questionar uma criança da era dos computadores. O seu mundo é dividido em duas partes: hardware, os componentes físicos que constituem um computador, e software, os programas que indicam ao computador o que fazer. Embora o hardware seja visível e tangível, a criança sabe que o software é a alma da máquina. Sem software, um computador é pouco mais do que um aglomerado de plástico e silicone sem qualquer utilidade. Um computador sem software é como um automóvel sem gasolina, uma máquina fotográfica sem filme, um aparelho estereofónico sem discos.” [Taylor, 1984], página 42.

Nas quase três décadas que decorreram desde a publicação deste artigo, verificaram-se avanços profundos tanto no “hardware” como no “software”. O “hardware” tornou-se mais rápido, mais barato, mais pequeno, a sua capacidade de armazenamento de informação aumentou ordens de magnitude, implantou-se em quase todos os aspectos relacionados com a vida moderna, telemóveis, agendas electrónicas, máquinas fotográficas, automóveis, aparelhos electrodomésticos, 389

CAPÍTULO 15. EPÍLOGO

entre milhares de outros. O “software” foi sujeito a novos desafios, abriram-se novas áreas de aplicação, aumentou-se a complexidade das aplicações, surgiu a Internet. No entanto, a ideia transmitida no artigo da TIME mantém-se, sem o “software”, os computadores (“hardware”) não têm utilidade.

Como foi bem caracterizado pela revista TIME, o “software” é o feiteiro dentro da máquina. Um dos objectivos da Informática corresponde ao estudo e desenvolvimento das entidades abstractas que residem nos computadores, os processos computacionais. Um processo computacional é um ente imaterial que evolui ao longo do tempo, executando acções que levam à solução de um problema. A evolução de um processo computacional é ditada por uma sequência de instruções a que se chama programa, e a actividade de desenvolvimento de programas é chamada programação.

Neste livro apresentámos conceitos básicos de programação. Neste último capítulo-

tulo fornecemos uma perspectiva dos principais conceitos introduzidos e apon-tamos direcções para o seu estudo mais aprofundado. Juntamente com cada tópico, fornecemos a indicação de disciplinas da Licenciatura em Engenharia Informática e de Computadores do Instituto Superior Técnico que abordam o tópico e referências bibliográficas nas quais o tópico é discutido com profundidade.

15.1

Programas

Um dos conceitos basilares da programação corresponde à noção de programa.

Um programa é uma entidade estática que define a evolução de um processo computacional.

Um programa relaciona-se com uma multiplicidade de outros conceitos: um programa corresponde a um algoritmo, um programa é expresso numa dada linguagem, um programa engloba um conjunto de abstracções, e a execução de um programa origina um processo computacional.

Na Figura 15.1 apresentamos as relações entre os conceitos associados com programa, indicando, dentro de um retângulo, os conceitos abordados neste livro.

15.1. PROGRAMAS

391

Matemática

Álgebra

Arquitetura

discreta

de computadores

Especificaç~

oes

Estruturas de

informaç~

ao

Controle

Processadores

de linguagens

Abstracç~

oes

Processo

Programa

Linguagem

computacional

Estudo de

linguagens

formais

Complexidade

Algoritmo

Teoria da

computacional

Computabilidade

computaç~

ao

Fundamentos

Figura 15.1: “Mapa” dos principais conceitos associados a programa.

15.1.1

Algoritmos

Um programa é caracterizado matematicamente como sendo um algoritmo –

uma sequência de passos que podem ser executados mecanicamente (isto é, sem exigir recurso à criatividade ou à imaginaç~ao), de modo a atingir um determinado objectivo.

Um algoritmo está sempre associado a um dado objectivo, ou seja, à soluç~ao de um dado problema. A execuç~ao das instruções do algoritmo garante que o seu objectivo é atingido. A sequência de passos de um algoritmo deve ser executada por um agente, o qual pode ser humano, mecânico, electrónico, ou qualquer outra coisa. Cada algoritmo está associado a um agente (ou, mais correctamente, a uma classe de agentes) que deve executar as suas instruções.

Aquilo que representa um algoritmo para um agente pode não o representar para outro agente.

Embora um algoritmo não seja mais do que uma descrição da sequência de passos a seguir para atingir um objectivo, nem todas as sequências de passos para atingir um objectivo podem ser

CAPÍTULO 15. EPÍLOGO

deve possuir três características: (1) um algoritmo é rigoroso; (2) um algoritmo é eficaz; (3) um algoritmo deve terminar.

O estudo de algoritmos é uma das principais vertentes ligadas à programação.

Este estudo pode ser encarado sob várias perspectivas:

— O estudo dos fundamentos dos algoritmos

Esta perspectiva aborda o conceito de computabilidade (ou seja, qual o significado da execução mecânica de instruções) e as limitações teóricas a que estão sujeitos os algoritmos.

Este estudo, fundamentalmente de natureza matemática, começou a ser abordado nos anos 30, antes do advento dos computadores digitais, com os trabalhos de Church, Kleene, Post e Turing, e continua a ser de grande relevância para a programação, pois permite-nos determinar até que ponto podemos garantir que existe um algoritmo para resolver um dado problema. O domínio científico que se debruça sobre este tema é conhecido por Teoria da Computação [Sernadas, 1993], [Sipser, 2012].

— O estudo da complexidade de algoritmos

No Capítulo 7 apresentámos alguns padrões típicos da evolução de processos computacionais, estudando a ordem de grandeza do número de operações associadas ao processo e o “espaço” exigido pela evolução global do processo. Vimos que, para resolver um mesmo problema, é possível desenvolver algoritmos diferentes, originando processos computacionais que apresentam grandes diferenças na taxa a que consomem recursos.

A disciplina de Análise e Síntese de Algoritmos aborda o estudo da complexidade computacional associada a diferentes algoritmos, a definição de várias medidas para avaliar a ordem de crescimento de processos (para além da notação O apresentada na Secção 5.7 existem muitas outras, por exemplo, a notação Θ , a notação o e a notação Ω) e a classificação de classes de “dificuldade” de problemas (por exemplo, problemas de dificuldade P e problemas de dificuldade

A análise de algoritmos necessita de um conjunto de conceitos matemáticos tais como análise combinatória, teoria das probabilidades, manipulação al-gébrica, as quais s~ao tratadas num ramo da matemática conhecido por Matemática Discreta [Graham et al., 1989], [Johnsonbaugh, 2009].

15.1. PROGRAMAS

393

15.1.2

Linguagens

De modo a expressar algoritmos é necessário utilizar linguagens formais (linguagens que não apresentam ambiguidade).

Qualquer linguagem apresenta dois aspectos distintos, a forma da linguagem e o significado associado à forma. Estes aspectos s~ao denominados, respectivamente, sintaxe e semântica da linguagem. A sintaxe de uma linguagem é o conjunto de regras que definem quais as relações válidas entre os componentes da linguagem, tais como nomes, expressões, formas. A sintaxe nada diz em relação ao significado das frases da linguagem. A semântica de uma linguagem define qual o significado de cada frase da linguagem. A semântica nada diz quanto ao processo de geração das frases da linguagem.

Uma linguagem que é compreendida por um computador é chamada uma linguagem de programação. Ao estudar uma linguagem de programação, é fundamental uma perfeita compreensão da sua sintaxe e da sua semântica: a sintaxe vai determinar a forma das instruções a fornecer ao computador, e a semântica vai determinar o que o computador faz ao executar cada uma das instruções.

É pois natural que o estudo de linguagens seja um aspecto importante associado à Programação. Este estudo pode ser abordado através de várias perspectivas:

— O estudo de linguagens formais

Aborda o estudo dos diferentes tipos de gramáticas, das linguagens que estas originam, e da relação de uma linguagem com o conceito de autómato, um modelo abstracto de uma “máquina” que é comandada por uma linguagem. Esta área que surge em meados dos anos 50 com

o trabalho de Noam Chomsky [Chomsky, 1956] tem tido uma influência profunda no desenvolvimento das linguagens de programação modernas. O estudo de linguagens formais é abordado no domínio científico da Teoria da Computação [Ginsburg, 1966], [Sernadas, 1993], [Moret, 1997].

— O estudo de processadores de linguagens

Aborda os métodos para o desenvolvimento de camadas de abstracção que

“entendam” determinadas linguagens de programação.

Antes de iniciar a discussão deste tópico, convém mencionar alguns dos tipos de linguagens de programação. As linguagens de programação utilizadas no

394

CAPÍTULO 15. EPÍLOGO

desenvolvimento da esmagadora maioria das aplicações são conhecidas por linguagens de alto nível.

Uma linguagem de alto nível apresenta semelhanças com as linguagens que os humanos usam para resolver problemas, e é independente do computador onde é executada. O Python é um exemplo de uma linguagem de alto nível.

As linguagens de alto nível diferem drasticamente das linguagens que na realidade comandam directamente o funcionamento dos computadores.

Sendo os computadores digitais, toda a informação neles representada recorre apenas a dois estados discretos (e daí a designação de digital). A linguagem máquina é a linguagem utilizada para comandar directamente as acções do computador. As instruções em linguagem máquina são constituídas por uma sequência de símbolos correspondendo às entidades discretas manipuladas pelo computador (normalmente representados por 0 e por 1) e actuam directamente sobre os componentes do computador. A linguagem máquina é difícil de usar e de compreender por humanos e varia de computador para computador (é a linguagem nativa de cada computador).

Para que os computadores possam “entender” os programas escritos

numa linguagem de alto nível, existem programas que “traduzem” as linguagens de alto nível em linguagem máquina (recorde-se, que no início do Capítulo 1, dissemos que um computador é uma “caixa electrónica” que tem a capacidade de compreender e de executar as instruções que correspondem a programas). Esta “caixa electrónica” corresponde a uma camada de abstracção que se chama máquina virtual – corresponde à realização de uma máquina conceptual através de um programa. Existem duas abordagens para fazer esta tradução, os interpretadores e os compiladores:

— Interpretadores

No caso do Python, a tradução é feita através de um programa, chamado interpretador do Python, que lê formas, avalia-as e fornece o seu resultado.

Programa

◆◆◆◆◆◆◆◆◆◆ Interpretador

Resultado da

execuç~

ao



★

★

Dados

Figura 15.2: Conceito de interpretação.

15.1. PROGRAMAS

395

Com a utilização de um interpretador, a máquina virtual recebe uma forma (instrução) de cada vez, analisa-a e executa a sequência de instruções que, em linguagem máquina, corresponde a essa forma. Só depois da forma avaliada, é que a máquina virtual vai receber a próxima forma (Figura 15.2). Esta abordagem dá origem ao ciclo lê-avalia-escreve discutido no início do Capítulo 2.

— Compiladores

No caso de outras linguagens de alto nível, por exemplo o C, a tradução é feita através de um programa, chamado compilador, que recebe um programa escrito na linguagem (a que se dá o nome de programa fonte) e que produz um programa equivalente (equivalente no sentido em que com os mesmos dados produz os mesmos resultados) escrito em linguagem máquina (a que se dá o nome de programa objecto).

Programa

Programa

*

*

Compilador



fonte

objecto

***** Máquina

* Resultado da

física ou

execuç~

ao

virtual



*

Dados

Figura 15.3: Conceito de compilação.

Com a utilização de um compilador, fornecem-se todas as instruções do programa fonte e, antes de o começar a executar, o compilador

analisa cada uma das suas instruções e substitui-a pelo conjunto de instruções que a realiza, criando-se assim um novo programa, o programa objecto, equivalente ao primeiro, mas constituído por instruções para outra “máquina”.

Em compilação, a execução de um programa passa por duas fases distintas: em primeiro lugar cria-se o programa objecto, depois executa-se o programa objecto (Figura 15.3).

Uma das vertentes associadas à programação corresponde ao desenvolvimento de interpretadores e de compiladores [Aho et al., 1986], ou, de um modo mais genérico, de processadores de linguagens, os quais correspondem à operacionalização de linguagens de programação. No desenvolvimento de um processador para uma linguagem, vários aspectos têm de ser considerados, tais como a análise léxica das frases da linguagem (o reconhecimento individual dos componentes), a análise sintáctica das frases, a análise semântica,

396

CAPÍTULO 15. EPÍLOGO

a criação de estruturas resultantes da análise, a sua tradução, a optimização do código gerado, os modelos de avaliação, a gestão de memória, etc.

O estudo de processadores de linguagens, para além de necessitar de conceitos associados ao estudo de linguagens formais, está intimamente ligado com o estudo da Arquitectura de Computadores [Mano e Kime, 1997], [Arroz et al., 2007], ou seja, o estudo dos vários componentes que constituem um computador, as suas interligações físicas e o modo como estes componentes comunicam entre si.

— O estudo comparativo de linguagens de programação Para dominar a programação, é essencial um conhecimento pormenorizado de várias linguagens de programação. O estudo comparativo de linguagens de programação [Fischer e Grodzinsky, 1993], [Wilson e Clark, 1988], [Scott, 2000], [Turbak et al., 2008] visa a análise do modo como diferentes linguagens abordam os conceitos subjacentes à programação.

Existem conceitos que estão presentes em todas as linguagens de programação, tais como tipos de informação, estruturas de controle, abstracção, nomeação, entre muitos outros. A análise da utilização destes conceitos em diferentes linguagens permite a sua compreensão

a um nível de abstracção mais elevado, favorecendo a sua utilização em linguagens específicas. Este estudo não só aumenta a compreensão sobre linguagens de programação, como também facilita a aprendizagem de novas linguagens.

15.1.3

Construção

de abstracções

ões

Uma técnica fundamental em programação, a abordagem do topo para a base, consiste em identificar os principais subproblemas que constituem o problema a resolver e em determinar qual a relação entre eles. Este processo é repetido para cada um dos subproblemas, até se atingirem problemas cuja solução é simples.

A abordagem do topo para a base permite estruturar um problema em subproblemas, de modo a perceber os passos que têm de ser seguidos para atingir a sua solução. Cada um dos subproblemas em que um problema é dividido corresponde a uma abstracção – existem pormenores que são ignorados, nomeadamente os passos para os resolver.

Qualquer programa corresponde a um algoritmo para resolver um modelo, ou

15.1. PROGRAMAS

397

uma abstracção, de um dado problema. Sob este ponto de vista, a actividade de programação corresponde a uma construção de abstracções. Estas abstracções estão estruturadas em camadas. Em cada camada de abstracção interessa separar o que é essencial ao problema dos conceitos ou pormenores acessórios. A abstracção está omnipresente em programação.

As primeiras abstracções utilizadas em programação corresponderam à abstracção de máquinas e à abstracção procedimental.

— Abstracção de máquinas

A abstracção de máquinas consiste na criação de máquinas virtuais

(como descrito na Secção 15.1.2) que aceitam uma dada linguagem de programação, permitindo abstrair a linguagem subjacente ao computador utilizado e os pormenores exigidos por esta linguagem.

Assim, uma linguagem de programação corresponde à abstracção de uma máquina virtual cuja “linguagem máquina” é a linguagem de programação

– o computador é assim a “caixa electrónica” que tem a capacidade de compreender e de executar as instruções que correspondem a programas.

— Abstracção procedimental

A abstracção procedimental consiste em abstrair do modo como as funções realizam as suas tarefas, concentrando-se apenas na tarefa que as funções realizam. Ou seja, corresponde à separação do “como” de “o que”. A abstracção procedimental é realizada através da nomeação de funções, abstraindo-se do modo como estas realizam a sua tarefa.

Posteriormente, reconheceu-se que num programa são necessárias outras duas facetas da abstracção, a abstracção de controle e a abstracção de dados. Esta verificação levou à famosa expressão introduzida por [Wirth, 1976]: “Algoritmos

+ Estruturas de Informação = Programas”.

— Abstracção de controle

A abstracção de controle corresponde à conceptualização de mecanismos que especificam o modo como as instruções de um programa são executadas.

Em linguagem máquina, o controle é realizado fundamentalmente através da sequenciação implícita, seguindo a sequência física das instruções de um

398

CAPÍTULO 15. EPÍLOGO

programa, complementada por dois mecanismos de excepção, o salto incondicional e o salto condicional. Estas estruturas de controle aparecem enraizadas nas primeiras linguagens de programação.

O movimento em favor da programação estruturada (liderado nos anos 60 por Edsger W. Dijkstra [Dijkstra, 1976] e [Dahl et al., 1972]) levou à introdução de mecanismos abstractos de controle, tais como o if-then-else, o case, o repeat-until, o while-do e o for, que permitem que o programador se concentre em especificar o controle da execução em termos de actividades simples, modulares e bem compreendidas.

Mais recentemente, o paralelismo permite conceptualizar a utilização de várias máquinas em simultâneo, abstraindo do facto que, eventualmente, a máquina subjacente apenas tem um processador.

— Abstracção de dados

A abstracção de dados surge nos anos 70 com o trabalho de [Liskof e Zilles, 1974], [Liskof e Guttag, 1986] e corresponde à separação entre o estudo das propriedades dos dados e os pormenores da realização dos dados numa linguagem de programação. A abstracção de dados é traduzida pela separação das partes do programa que lidam com o modo como os dados são utilizados das partes que lidam com o modo como os dados são representados.

A abstracção de dados, corporizada na teoria dos tipos abstractos de informação, tem grandes ligações com a matemática discreta e as álgebras.

Juntamente com o desenvolvimento da abstracção de dados, têm surgido novas abstrações de controle, associadas a tipos específicos de dados, por exemplo, os iteradores e os geradores.

Antes de concluir esta secção, convém referir que a distinção entre programas e dados corresponde também a uma abstracção. De facto, considerando a Figura 15.3, ao utilizarmos um compilador, o programa fonte para o programador é um programa; no entanto, o mesmo programa, para o compilador, corresponde a dados.

15.2

Programação

ao

De um modo abstracto, o desenvolvimento de um programa pode ser visto como uma sequência de fases através das quais as descrições de um sistema se tornam

progressivamente mais pormenorizadas. Começando com a análise do problema, que dá ênfase ao que tem de ser feito, a descrição é refinada progressivamente para a descrição de como o problema é resolvido de um modo mecânico. Para isso, na fase do desenvolvimento da solução descreve-se rigorosamente como o problema vai ser resolvido, sem se entrar, no entanto, nos pormenores inerentes a uma linguagem de programação. Na fase da programação da solução o algoritmo desenvolvido é escrito recorrendo a uma linguagem de programação. Em resumo, durante a sequência de fases seguida no desenvolvimento de um programa, a caracterização de o que tem de ser feito transforma-se progressivamente numa especificação de como vai ser feito.

As fases do desenvolvimento de programas que levam dos requisitos iniciais ao desenvolvimento de código executável são guiadas por metodologias adequadas e sistemáticas, estudadas na disciplina de Engenharia de Software, e descritas, por exemplo, em [Alagar e Periyasamy, 1998], [Jalote, 1997], [Sommerville, 1996],

[Pfleeger e Atlee, 2010].

Para além das metodologias de desenvolvimento de programas, a programação estuda a arquitectura dos programas, utiliza técnicas desenvolvidas em vários domínios da Informática e pode ser abordada seguindo diversos paradigmas.

A programação pode abordar o desenvolvimento de aplicações concretas ou o desenvolvimento de programas para gerir outros programas, como é o caso dos sistemas operativos.

Na Figura 15.4 apresentamos as relações entre os conceitos associados com programação, indicando, dentro de um rectângulo, os conceitos tratados neste livro.

15.2.1

Arquitectura de programas

Um programa complexo é constituído por vários componentes que comunicam entre si. Estes componentes podem ser módulos do próprio programa ou podem ser outros programas existentes na organização para a qual o programa é desenvolvido.

A Arquitectura de Programas (do inglês “Software Architecture”) aborda os vários métodos para estruturar componentes e os mecanismos possíveis para a sua comunicação [Bass et al., 1998], [Hofmeister et al., 2000].

A arquitectura de programas é uma abstracção que ignora os algoritmos e a

400

CAPÍTULO 15. EPÍLOGO

Arquitectura de

programas

Inteligência

Programaç~

ao imperativa

artificial

Programaç~

ao funcional

Interfaces

Técnicas

Paradigmas

gráficas

Programaç~

ao

Programaç~

ao com objectos

Bases de dados

Programaç~

ao em lógica

Sistemas operativos

Arquitectura de computadores

Figura 15.4: “Mapa” dos principais conceitos associados a
programaç~ao.

representaç~ao dos dados e que se concentra no tipo de tarefas que
v~ao ser realizadas pelos programas, na partilha e na reutilizaç~ao de
componentes (componentes esses que podem ser escritos em diferentes
linguagens de programaç~ao), na interligaç~ao entre os componentes,
nos modos de comunicaç~ao e na localizaç~ao “geográfica” dos
componentes (estes podem ser sistemas centralizados ou sistemas
distribu´idos [Marques e Guedes, 1998]).

15.2.2

Paradigmas de programaç~

ao

Um paradigma de programaç~ao é um modelo para abordar o modo de
raciocinar durante a fase de programaç~ao e, consequentemente, o
modo como os programas s~ao escritos. Existem vários paradigmas de
programaç~ao, sendo os mais importantes a programaç~ao imperativa,
a programaç~ao funcional, a programaç~ao com objectos e a
programaç~ao em lógica.

— Programaç~ao imperativa

O paradigma da programaç~ao imperativa corresponde à abordagem
mais comum à programaç~ao. A programaç~ao imperativa segue o
modelo de computaç~ao designado por modelo de von Neumann¹ ou
modelo do programa ¹Em honra do matemático John von Neumann
(1903–1957) que teve grande influência no desenvolvimento do
conceito de computaç~

ao baseada num programa armazenado.

15.2. PROGRAMAC

armazenado – o qual está subjacente à esmagadora maioria dos computadores –, segundo o qual, tanto o programa como os dados estão armazenados na memória do computador e, durante a execução do programa, as células (ou posições) da memória são repetidamente acedidas, interpretadas e actualizadas.

A programação imperativa reflecte as acções executadas por um computador ao nível da linguagem máquina. A este nível, as principais acções executadas por um computador correspondem a ir buscar os conteúdos de certas posições de memória (que contêm as instruções – indicação de operadores e eventualmente de operandos), à aplicação da operação explicitada na instrução a valores obtidos da memória e em armazenar o resultado numa determinada posição de memória. As linguagens imperativas abstraem a noção de conteúdo de uma posição de memória, substituindo-a pelo valor de uma variável, mas os passos seguidos são basicamente os mesmos.

A base da programação imperativa reside na operação de atribuição através da qual uma variável recebe o resultado de uma operação – como vimos no Capítulo 5, isto corresponde à utilização de efeitos. Paralelamente, a programação imperativa recorre exaustivamente a métodos para controlar e especificar a execução de sequências de instruções. A tarefa de programação é considerada como a escrita de uma sequência de ordens dadas ao computador (e daí a designação de imperativa).

São exemplos de linguagens baseadas na programação imperativa o C, o Pascal, o Fortran e o Basic.

— Programação funcional

A programação funcional [Cousineau e Mauny, 1998], [Hudac, 1989], [MacLennan, 1990] tem as suas raízes no modelo desenvolvido pelo matemático Alonzo Church (1903–1995) sobre a teoria das funções recursivas e sobre o cálculo lambda. A programação funcional é baseada no conceito de função matemática, uma transformação de elementos do domínio em elementos do contradomínio. O paradigma funcional considera os programas como funções matemáticas – caixas pretas que recebem um certo número

de valores de entrada, os parâmetros concretos, e produzem um único valor, o resultado. No paradigma funcional, o único modo de especificar as acções de uma função é através da invocação de outras funções.

Uma das propriedades das linguagens funcionais corresponde à não existência

402

CAPÍTULO 15. EPÍLOGO

de variáveis e à não existência da operação de atribuição. Os parâmetros de funções, embora sejam nomeados e utilizados de modo semelhante a variáveis, nunca são alterados. Sob esta perspectiva, os parâmetros de funções podem ser considerados como designações de constantes.

Um outro aspecto associado à programação funcional é o facto de os parâmetros de funções e o valor de funções poderem ser outras funções. Este aspecto faz com que frequentemente se diga que a programação funcional trata as funções como cidadãos de primeira classe.

O principal mecanismo de controle da execução em linguagens funcionais corresponde à recursão combinada com a utilização de expressões condicionais (de que é exemplo a expressão `cond` em Scheme), as quais avaliam condições (expressões cujo valor é verdadeiro ou falso) e, mediante o resultado da avaliação, avaliam selectivamente outras funções.

São exemplos de linguagens exclusivamente baseadas no paradigma da programação funcional o Miranda, o Haskell e o FP. O Scheme, o Lisp e o ML

são linguagens que, embora baseadas na programação funcional, apresentam algumas características imperativas.

— Programação com objectos

O paradigma de programação com objectos [Meyer, 1997], [Schach, 2005]

surge com o desenvolvimento das ideias introduzidas, em meados dos anos 60, pela linguagem Simula, juntamente com o reforço dos conceitos ligados à abstracção de dados.

A programação com objectos, embora baseada no modelo de von Neumann, difere da programação imperativa, pois em vez de considerar um processador e uma memória monolíticos, considera um modelo de computação mais estruturado e distribuído, organizando os processos computacionais em torno de objectos, cada um dos quais pode ser considerado um “computador” independente com a sua própria memória (o seu estado) e as suas funções executáveis (os métodos) que manipulam a sua memória.

Em programação com objectos, a programação é centrada na construção de abstrações chamadas objectos, os quais comunicam entre si através de mensagens e estão organizados em estruturas hierárquicas de classes. Associado à organização de classes, existe um mecanismo de herança, o qual permite a criação de novas abstrações através da especialização e da combinação das classes existentes.

15.2. PROGRAMAC

~
,

AO

403

São exemplos de linguagens baseadas no paradigma da programação com objectos, o Smalltalk, o Eiffel, o C + +, o Clos (Common Lisp Object System), o Java e o Python.

— Programação em lógica

O paradigma da programação em lógica [Hogger, 1990] tem as suas raízes no desenvolvimento da lógica matemática e nos trabalhos de Herbrand [Herbrand, 1930] e de Robinson [Robinson, 1965] sobre a demonstração automática de teoremas. De um modo simples (e ideal) podemos definir a tarefa da programação em lógica do seguinte modo: O programador descreve as propriedades lógicas que caracterizam o problema a resolver. Esta descrição é utilizada pelo sistema para encontrar uma solução do problema a resolver (ou para inferir a solução para o problema).

A abordagem da programação em lógica é obtida representando as especificações que caracterizam o problema num componente individualizado, que se chama base de conhecimento. A base de conhecimento é independente do programa que a manipula, programa esse que é chamado a máquina de inferência. Em programação em

lógica, as descrições de um problema são fornecidas num formalismo baseado em lógica de primeira ordem.

Suponhamos, a título de exemplo, que desejávamos especificar o efeito de procurar um elemento, x , numa lista de elementos, L . Podemos definir o predicado em? como verdadeiro, sempre que o elemento x pertence à lista L .

Este predicado pode ser especificado do seguinte modo: Para qualquer elemento x e qualquer lista L : $\text{em?}(x, L)$ se e só se

$L =$ lista que apenas contém x

ou

$L =$ junção de L_1 e L_2 e $(\text{em?}(x, L_1) \text{ ou } \text{em?}(x, L_2))$ esta especificação descreve uma função de procura de um modo declarativo: o elemento pertence à lista se a lista é constituída exactamente por esse elemento; em caso contrário consideramos a lista como sendo constituída por duas sublistas cuja junção é a lista original, estando o elemento na lista se este estiver numa das sublistas.

A programação em lógica baseia-se no facto de que existem duas abordagens distintas para representar o conhecimento: através de declarações (a repre-

404

CAPÍTULO 15. EPÍLOGO

sentação declarativa) e através de funções (a representação procedimental).

A representação declarativa parte do princípio de que saber é “saber o quê”

e, consequentemente, o conhecimento é representado através de um conjunto de declarações ou afirmações descrevendo um domínio particular. Sobre estas declarações actua uma função de ordem geral que permite inferir nova informação. A representação procedimental parte do princípio de que saber é “saber como” e, consequentemente, o conhecimento é representado como um conjunto de funções, cada uma das quais representa um fragmento de conhecimento.

Do ponto vista formal não existe distinção entre estas duas abordagens. Sabemos que os programas podem ser olhados como

dados, e sob este ponto de vista tudo é declarativo. Por outro lado, podemos olhar para as declarações como instruções de um programa, e sob este ponto de vista tudo são funções.

O paradigma da programação em lógica é corporizado na linguagem Prolog.

Esta linguagem foi desenvolvida em 1972 associada à resolução de problemas em Inteligência Artificial. Actualmente, o Prolog apresenta um grande leque de aplicações que transcendem a Inteligência Artificial. O Prolog tem sido utilizado para a definição de tradutores de linguagens, interpretadores e compiladores, bases de dados dedutivas, interfaces em língua natural, *etc.*

A programação em lógica apenas satisfaz parcialmente a abordagem declarativa. Para tornar a execução do programa eficiente, é introduzido um certo número de compromissos que diluem o aspecto da representação declarativa pura. Isto significa que o programador tem de se preocupar com mais aspectos do que apenas a especificação daquilo que o programa é suposto fazer.

Neste sentido, existem aspectos não declarativos que podem ser vistos como indicações dadas pelo programador à máquina de inferência. De um modo geral, estes aspectos reduzem a clareza das descrições do programa, pois misturam a descrição do problema com preocupações de implementação.

Usando a programação em lógica, um programa é constituído por uma série de declarações em lógica, juntamente com um conjunto de indicações pro-cedimentais que controlam a utilização das declarações. Por esta razão um programa é considerado como uma combinação de lógica e de controlo, traduzidas na conhecida expressão: “Programa = Lógica + Controlo” [Kowalski, 1979].

15.2. PROGRAMAC

~
s

AO

405

15.2.3

Técnicas usadas em programação

Existem vários domínios científicos em Informática que contribuem para a programação com um conjunto de técnicas que estendem e que facilitam a utilização dos computadores. Nesta secção apresentamos resumidamente três desses domínios científicos, a Inteligência Artificial, as interfaces gráficas e o domínio associado ao desenvolvimento de bases de dados.

— Inteligência Artificial

A Inteligência Artificial [Russell e Norvig, 2010] é o ramo da Informática que estuda os métodos para os computadores executarem tarefas que, quando executadas por humanos, são ditas requerer inteligência.

Embora a Informática tenha contribuído para o desenvolvimento de programas que resolvem problemas muito variados, estes problemas têm uma característica em comum, a de a sua solução poder ser descrita por modelos matemáticos e, portanto, poder ser expressa por um algoritmo. A solução da maioria dos problemas que encaramos na nossa vida quotidiana não pode ser descrita por um algoritmo – existe falta de especificação completa do problema, existem ambiguidades e, por vezes, mesmo contradições. A nossa capacidade para encarar e resolver com sucesso problemas deste tipo dá-nos o direito à classificação de inteligentes. Para resolver estes problemas, nem sempre seguimos regras exactas, frequentemente utilizamos regras empíricas, acumuladas ao longo de anos de experiência que, embora não nos garantam a solução dos problemas, a maior parte das vezes resultam. Estas regras empíricas são chamadas heurísticas.

Existe ainda outro tipo de problemas para os quais, embora a solução possa ser descrita por algoritmos, estes não são úteis devido à sua complexidade computacional. Por mais potentes que sejam os computadores, os problemas deste tipo não podem ser resolvidos algoritmicamente, devido a limitações de espaço e de tempo. Uma vez mais, são as heurísticas que permitem a resolução destes problemas. Embora não garantam a melhor solução, e por vezes nem sequer solução alguma, as heurísticas permitem, a maior parte das vezes, resolvê-los num período de tempo razoável.

Uma das características das técnicas usadas em Inteligência Artificial é a orientação para problemas que podem ser expressos em termos simbólicos. O

2A palavra heurística vem da palavra grega heuriskein que significa descobrir.

406

CAPÍTULO 15. EPÍLOGO

processamento simbólico engloba um conjunto de metodologias e de técnicas que foram criadas para resolver problemas não estruturados, que lidam com informação não rigorosa e incompleta, e que usam regras empíricas ganhas com a experiência. O processamento simbólico consiste na utilização de informação ou de conhecimento representado através de símbolos. Os símbolos são usados para representar objectos, situações, acontecimentos do mundo real e as propriedades a elas associadas. Os símbolos podem ser interligados, usando estruturas como grafos ou redes, para representar relações como hierarquias e dependências.

Uma outra característica da Inteligência Artificial é o recurso a heurísticas.

Uma heurística é qualquer função para a resolução de um problema que não é um algoritmo ou que não foi demonstrado ser um algoritmo. As razões mais comuns para que uma sequência de instruções seja uma heurística e não um algoritmo são: (1) a execução das instruções não termina para algumas classes de problemas; (2) não se garante que produza uma situação em que o objectivo tenha sido atingido, devido a problemas com a própria heurística; ou (3) não se garante que produza uma situação em que o objectivo tenha sido atingido, por o problema que pretende resolver não ser bem definido.

A Inteligência Artificial estuda também técnicas para incorporar formas de raciocínio em máquinas. O raciocínio é feito através de um processo de inferência. Recorrendo a técnicas de raciocínio, a Inteligência Artificial permite um tipo diferente de programação: em vez de especificar o comportamento a atingir pelo sistema em função de sequências e repetições de acções, o programador fornece um conjunto de regras lógicas, semelhantes às usadas em actividades quotidianas. A responsabilidade de ligar essas regras de um modo lógico, com a finalidade de obter o comportamento desejado, é da exclusiva responsabilidade do componente do sistema encarregado do raciocínio. A independência das regras em relação ao componente do programa que sabe como usar essas regras é uma das características que distingue muitos dos sistemas de Inteligência Artificial. Esta

separação é mantida para que o mesmo programa possa ser usado, mesmo quando um novo domínio requiera um novo conjunto de regras. Esta independência garante que, quando introduzimos uma nova regra, não tenhamos de pensar em todas as situações em que ela vai ser usada e como vai ser usada.

Com a Inteligência Artificial reconheceu-se a necessidade de utilizar conhecimento para se obter comportamento inteligente. Por outras palavras, não

15.2. PROGRAMAC

~
,

AO

407

basta possuir capacidade de raciocínio, é necessária uma grande quantidade de conhecimento para permitir que o raciocínio possa resolver problemas.

A Inteligência Artificial desenvolve técnicas para representar, manipular e explicitar o conhecimento.

— Interfaces Gráficas

As técnicas de interfaces gráficas [Foley et al., 1997] têm sido tornadas possíveis devido ao desenvolvimento de novas tecnologias associadas à construção de unidades de visualização de informação (os monitores de computador), ao aumento da velocidade de processamento dos computadores e ao desenvolvimento de um conjunto de algoritmos que permite a manipulação e a visualização das entidades envolvidas.

O domínio das interfaces gráficas preocupa-se com a construção de abstrações de objectos e com a sua apresentação sob a forma de figuras. As interfaces gráficas estão hoje vulgarizadas através dos sistemas operativos do Macintosh e do Windows, nos quais a interacção entre o computador e o utilizador recorre à manipulação interactiva de ícones e de janelas. Nos últimos anos, o tratamento digital de imagens e a crescente utilização de técnicas de animação tem vulgarizado as aplicações das interfaces gráficas.

As interfaces gráficas, de um modo genérico, lidam com a síntese, em forma de figuras, de objectos reais ou imaginários. Recentemente, a

utilizaç~ao conjunta de documentos contendo texto, gráfcos e imagens (conhecida por multimédia) está em franco desenvolvimento.

As técnicas de interfaces gráficas podem ser classificadas segundo vários aspectos:

— As dimens~oes utilizadas na representaç~ao do objecto (objectos bidimensi-onais ou objectos tridimensionais) e o tipo de imagem a ser produzida (a preto e branco, em tonalidades de cinzento ou com cor).

— O tipo de interacç~ao com a imagem, o qual define qual o grau de controle que o utilizador tem sobre a imagem ou sobre o objecto.

— O papel da imagem em relaç~ao à sua utilizaç~ao, ou seja, se a visualizaç~ao da imagem corresponde ao objectivo final ou se corresponde a um mero meio para atingir outro objectivo.

— As relaç~oes entre os objectos e as suas imagens, as quais apresentam uma gama extensa de possibilidades, desde a utilizaç~ao de uma única imagem

408

CAPÍTULO 15. EPÍLOGO

estática, à utilizaç~ao de várias imagens que evoluem e interagem ao longo do tempo.

A tendência actual em interfaces gráficas concentra-se também na modelaç~ao dos objectos, não apenas na criaç~ao das suas imagens.

— Bases de Dados

As tecnologias de bases de dados são fundamentais para os sistemas de informaç~ao existentes em organizaç~oes. De um modo simples, uma base de dados corresponde a um repositório de informaç~ao, juntamente com um conjunto de programas, o sistema de gest~ao da base de dados, que acede, guarda e gere a informaç~ao contida na base de dados [Ramakrishnan e Gehrke, 2000].

Sob esta perspectiva, uma base de dados parece ser semelhante a um tipo abstracto de informaç~ao.

Uma base de dados pode ser considerada a três n´iveis de abstracç~ao, o n´ivel f´isico, o n´ivel lógico e o n´ivel da interface.

— O n´ivel f´isico

Preocupa-se com a representação interna e com a organização da informação contida na base de dados. Este nível corresponde ao nível da representação de um tipo abstracto de informação. A este nível surgem diferentes preocupações associadas à programação: a gestão de memória aborda a utilização eficiente de enormes quantidades de informação, informação essa que é demasiado volumosa para ser totalmente carregada na memória do computador; a gestão da concorrência aborda a partilha da informação entre vários utilizadores em simultâneo, garantindo a coerência dos dados na base de dados; a recuperação desenvolve técnicas para garantir a salvaguarda da informação no caso de ocorrência de erros (quer por parte do programa que manipula a base de dados, quer por parte do utilizador), falhas no funcionamento do computador (por exemplo, originadas por quebras de corrente), ou utilização maliciosa.

— O nível lógico

Desenvolve modelos de dados que correspondem à organização conceptual da informação na base de dados. Destes modelos, o mais vulgarizado é o modelo relacional através do qual uma base de dados é considerada como um conjunto de tabelas, com eventuais relações entre si.

O nível lógico apresenta uma linguagem para a definição dos aspectos estruturais da informação (a linguagem de definição de dados) e uma

15.2. PROGRAMAC

, ~

AO

409

linguagem para aceder à informação contida na base de dados e para efectuar a actualização desta informação (a linguagem de manipulação de dados). Com estas linguagens desenvolvem-se programas específicos para a manipulação de bases de dados.

Certos aspectos tratados ao nível físico, como por exemplo, a gestão da concorrência, traduzem-se ao nível lógico em outros conceitos, por exemplo, o conceito de transacção.

— O nível da interface

Aborda a organização e apresentação da informação contida na base de dados. Este nível permite criar camadas de abstracção sobre a informação ao nível lógico, de modo a que diferentes utilizadores possam ter percepções diferentes do nível de granularidade da informação contida na base de dados.

As bases de dados são uma das tecnologias em que muito trabalho está a ser realizado, tendo surgido nos últimos anos bases de dados de objectos que combinam a programação com objectos com a utilização de bases de dados e as bases de dados dedutivas que combinam as capacidades de raciocínio desenvolvidas em Inteligência Artificial com o armazenamento de grandes quantidades de informação.

15.2.4

Sistemas operativos

O sistema operativo de um computador [Marques e Guedes, 1994], [Silberschatz et al., 2001], [Tanenbaum, 2001] é um programa que controla a execução de todos os programas dentro do computador – funciona como um supervisor de todas as operações efectuadas pelo computador.

Um sistema operativo corresponde a uma máquina virtual que esconde o funcionamento físico dos vários componentes e unidades ligadas a um computador –

processador, memória, unidades de disco, impressoras, rede, *etc.*

Ao nível do sistema operativo, o utilizador fornece indicações tais como, “execute este programa”, “guarde esta informação”, “imprima esta informação”; cada uma destas instruções vai originar a execução de um programa, ao nível do sistema operativo, que efectua a acção necessária.

O sistema operativo tem várias funções específicas, entre as quais:

410

CAPÍTULO 15. EPÍLOGO

— Criar, manter e permitir o acesso ao sistema de ficheiros. Um ficheiro pode ser definido como um conjunto de informação – programas ou dados. O

sistema operativo permite abstrair o modo como os ficheiros s~ao armazenados internamente, fornecendo a possibilidade de nomear ficheiros e organizá-los em estruturas hierárquicas.

— Movimentar ficheiros entre as várias unidades de um computador, por exemplo, memória, disco e impressoras.

— Garantir a segurança de ficheiros através de um sistema de autenticaç~ao e de permiss~oes.

— Permitir o funcionamento do computador em regime de multitarefa (do inglês

“multi-tasking”). Este funcionamento permite que um computador realize várias tarefas em “simultâneo”, por exemplo, aceder à Internet, imprimir um ficheiro e executar um programa. Este funcionamento tira partido da grande velocidade de execuç~ao dos computadores e consiste em dividir a “atenç~ao”

do processador sequencialmente e repetitivamente por várias tarefas, dando a ideia ao utilizador de que estas est~ao a ser realizadas em simultâneo.

— Permitir a utilizaç~ao simultânea do mesmo computador por vários utilizadores, a chamada multiutilizaç~ao.

O desenvolvimento de sistemas operativos, que tem naturalmente ligaç~oes profundas com o estudo da Arquitectura de Computadores, aborda as seguintes grandes áreas:

— Gest~ao de processos

Um sistema operativo lida com vários processos computacionais, simultaneamente activos. A gest~ao dos vários processos envolve a possibilidade de interrupç~ao temporária da execuç~ao de um processo e posteriormente a activaç~ao desse processo a partir do ponto de interrupç~ao. Para isso, o sistema operativo, para cada processo, tem de manter e gerir um conjunto de variáveis de estado (semelhantes às descritas na Secç~ao 7.2).

Ao lidar com vários processos surgem vários aspectos que têm de ser considerados, o escalonamento e despacho de processos que aborda a decis~ao de qual o processo a que deve ser dado o direito de execuç~ao e qual o tempo de execuç~ao antes da próxima interrupç~ao, a sincronizaç~ao entre processos,

que aborda a comunicação entre os processos e o tratamento de eventuais problemas de bloqueamento (em inglês, “deadlocks”) entre processos nos quais dois ou mais processos comunicantes entre si podem ficar mutuamente bloqueados à espera que certo acontecimento se verifique.

— Gestão de memória

O sistema operativo deve gerir a informação que está na memória do computador e providenciar o tráfego de informação entre a memória e o disco do computador (ou outras unidades de armazenamento).

Para além disso, o sistema operativo lida com o conceito de memória virtual, um modo de simular que o computador tem uma memória superior à sua memória física, através do recurso ao espaço existente em disco.

— Sistema de ficheiros

O sistema operativo tem de manter e gerir o sistema de ficheiros, uma camada de abstracção construída sobre a informação contida no disco (ou em outras unidades de armazenamento). O sistema de ficheiros deve contemplar a possibilidade da existência de informação distribuída por vários computadores ligados em rede.

Associados à utilização do sistema de ficheiros existem todos os problemas de protecção e segurança da informação.

— Gestão de comunicações

O sistema operativo tem a responsabilidade da transferência da informação que é lida ou escrita por um programa, escondendo os pormenores físicos da unidade utilizada para a leitura ou escrita.

Este livro corresponde ao primeiro passo no mundo fascinante da programação.

Muitos outros passos se devem seguir, aprofundando os conceitos aqui apresentados e aprendendo muitos outros. Paralelamente com o

aprofundar dos conceitos de programação, é essencial programar, programar muito.

A utilização de várias linguagens de programação, pertencentes a vários paradigmas, é fundamental para dominar em pleno os conceitos apresentados neste livro.

412

CAPÍTULO 15. EPÍLOGO

Durante a aprendizagem de novas linguagens deve ser dada especial atenção ao modo como cada uma delas aborda os conceitos de programação, subindo acima dos pormenores sintáticos associados a cada linguagem.

Nos cerca de 60 anos da vida da Informática, largas centenas de linguagens de programação foram desenvolvidas. Algumas, frutos de modas e caprichos, tiveram uma vida muito curta, outras sobreviveram dezenas de anos. Seja qual for a evolução futura das linguagens de programação, os conceitos subjacentes às linguagens persistem. Daí a sua grande importância neste mundo em constante evolução que é a programação.

Bibliografia

H. Abelson, G. Sussman, e J. Sussman. Structure and Interpretation of Computer Programs. MIT Press, Cambridge, MA, 2 edição, 1996.

ACM. Computing curricula 2001. Technical report, Interim Review Task Force, Association for Computing Machinery, 2000.
www.acm.org/education/curric_vols/cc2001.pdf.

ACM.

Computer

science

curriculum

2008:

An

interim

revision

of

CS

2001.

Technical

report,

Interim

Review

Task

Force,

Association

for

Computing

Machinery,

2008.

www.computer.org/portal/cms/docs/ieeecs/ieeecs/education/cc2001/ComputerScience2008.pdf.

ACM. Computer science curricula 2013 (strawman draft). Technical report, Interim Review Task Force, Association for Computing Machinery and IEEE

Computer Society, 2012. <http://ai.stanford.edu/users/sahami/CS2013>.

A. Aho, R. Sethi, e J. Ullman. Compilers: Principles, Techniques and Tools.

Addison-Wesley Publishing Co., Reading, MA, 1986.

V. S. Alagar e K. Periyasamy. Specifications of Software Systems. Springer-Verlag, Heidelberg, Alemanha, 1998.

G. Arroz, J. Monteiro, e A. Oliveira. *Arquitectura de Computadores*. IST Press, Lisboa, Portugal, 2007.

U. Ascher e C. Greif. *A First Course in Numerical Methods*. SIAM, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2011.

L. Bass, P. Clements, e R. Kazman. *Software Architecture in Practice*. Addison-Wesley Publishing Co., Reading, MA, 1998.

413

414

BIBLIOGRAFIA

N. L. Biggs. The roots of combinatorics. *Historia Mathematica*, 6:109–136, 1979.

B. W. Boehm. *Software Engineering Economics*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1981.

C. Boyer. *História da Matemática*. Edgard Blucher Lda., São Paulo, S.P., 1974.

W. Brainerd e L. Landwebber. *Theory of Computation*. John Wiley & Sons, New York, N.Y., 1974.

F. Brooks. *The Mythical Man-Month*. Addison-Wesley Publishing Co., Reading, MA, 1975.

N. Chomsky. Three models for the description of language. *IRE Trans. Infor.*

Theory, 2(3):113–124, 1956.

N. Chomsky. *Syntactic Structures*. Mouton, The Hague, The Netherlands, 1957.

N. Chomsky. On certain formal properties of grammars. *Inf. and Control*, 2: 137–167, 1959.

A. Church. *The Calculi of Lambda Conversion*. *Annals of Mathematics Studies*.

Princeton University Press, Princeton, N.J., 1941.

T. H. Cormen, C. E. Leiserson, e R. L. Rivest. Introduction to Algorithms. The MIT Press, Cambridge, MA, 3 edição, 2009.

G. Cousineau e M. Mauny. The Functional Approach to Programming. Cambridge University Press, Cambridge, U.K., 1998.

O. Dahl e K. Nygaard. Simula – an algol-based simulation language. Comm. of the ACM, 9(9):671–678, 1967.

O. Dahl, E. Dijkstra, e C. Hoare. Structured Programming. Academic Press, New York, N.Y., 1972.

N. Dale e H.M. Walker. Abstract Data Types: Specifications, Implementations, and Applications. D. C. Heath and Company, Lexington, MA, 1996.

M. Davis. O Computador Universal. Editorial Bizâncio, Lisboa, Portugal, 2004.

E. Dijkstra. A Discipline of Programming. Prentice-Hall Inc., Englewood Cliffs, N.J., 1976.

BIBLIOGRAFIA

415

E.W. Dijkstra. A note on two problems in connexion with graphs. Numerische Mathematik, 1(1):269–271, 1959.

J. Edmonds. How to Think about Algorithms. Cambridge University Press, New York, N.Y., 2008.

A. E. Fischer e F. S. Grodzinsky. The Anatomy of Programming Languages.

Prentice-Hall Inc., Englewood Cliffs, N.J., 1993.

J. D. Foley, A. van Dam, S. K. Feiner, e J. F. Huges. Computer Graphics: Principles and Practice. Addison-Wesley Publishing Co., Reading, MA, 2

edição, 1997.

S. Ginsburg. The Mathematical Theory of Context Free Languages. McGraw-Hill Book Company, New York, N.Y., 1966.

R. L. Graham, D. Knuth, e O. Patashnik. Concrete Mathematics: A

Foundation for Computer Science. Addison-Wesley Publishing Co., Reading, MA, 1989.

F. Hennie. Introduction to Computability. Addison-Wesley Publishing Co., Reading, MA, 1977.

J. Herbrand. Recherches sur la théorie de la démonstration. PhD thesis, Sor-bonne, Université de Paris, Paris, France, 1930.

H. Hermes. Enumerability, Decidability, Computability. Springer-Verlag, New York, N.Y., 2 edic~ao, 1969.

C. Hoare. Notes on data structuring. In Dahl, Dijkstra, e Hoare, editors, Structured Programming. Academic Press, New York, N.Y., 1972.

C. Hofmeister, R. Nord, e D. Soni. Applied Software Architecture. Addison-Wesley Publishing Co., Reading, MA, 2000.

D.R. Hofstadter. Gödel, Escher, Bach: An Eternal Golden Braid. Basic Books, New York, N.Y., 1979.

D.R. Hofstadter. Gödel, Escher, Bach: Um Entrelaçamento de Gênios Brilhan-tes. Imprensa Oficial SP, Sao Paulo, S.P., 2011.

C. G. Hogger. Essentials of Logic Programming. Basil Blackwell Inc., Oxford, UK, 1990.

416

BIBLIOGRAFIA

J. Hopcroft e J. Ullman. Formal Languages and Their Relation to Automata.

Addison-Wesley Publishing Co., Reading, MA, 1969.

P. Hudac. Conception, evolution, and application of functional programming languages. ACM Computing Surveys, 21(3):359–411, 1989.

P. Jalote. An Integrated Approach to Software Engineering. Springer-Verlag, New York, N.Y., 2 edic~ao, 1997.

R. Johnsonbaugh. Discrete Mathematics. Prentice Hall, Englewood Cliffs, N.J., 7 edic~ao, 2009.

- S. Kleene. Introduction to Meta-mathematics. American Elsevier Publishing Co., New York, N.Y., 1975.
- M. Kline. Mathematical Thought: From Ancient to Modern Times. Oxford University Press, Oxford, U.K., 1972.
- D. Knuth. The Art of Computer Programming, Vol. 1, Fundamental Algorithms.
Addison-Wesley Publishing Co., Reading, MA, 1973a.
- D. Knuth. The Art of Computer Programming, Vol. 3, Sorting and Searching.
Addison-Wesley Publishing Co., Reading, MA, 1973b.
- E. Koffmann, P. Miller, e C. Wardle. Recommended curriculum for cs1, 1984.
Communications of the ACM, 27(10):998–1001, 1984.
- E. Koffmann, P. Miller, e C. Wardle. Recommended curriculum for cs2, 1984.
Communications of the ACM, 28(8):815–818, 1985.
- R. A. Kowalski. Algorithm = logic + control. Comm. of the ACM, 22(7): 424–436, 1979.
- H. Ledgard. Programming Proverbs. Hayden Book Co., New York, N.Y., 1975.
- W. Lenhart e M. Ringle. Strategies for Natural Language Processing. Lawrence Erlbaum Associates, Hillsdale, N.J., 1982.
- B. Liskof e J. Guttag. Abstraction and Specification in Program Development.
MIT Press, Cambridge, MA, 1986.
- B. Liskof e S. Zilles. Programming with abstract data types. SIGPLAN Symp.
on Very High Level Languages – SIGPLAN Notices, 9(4):50–59, 1974.

BIBLIOGRAFIA

M. Machtey e P. Young. An Introduction to the General Theory of Algorithms.

North Holland, New York, N.Y., 1978.

B. J. MacLennan. Functional Programming: Practice and Theory. Addison-Wesley Publishing Co., Reading, MA, 1990.

Z. Manna e R. Waldinger. Logical Basis for Computer Programming, Volume 1: Deductive Reasoning. Addison-Wesley Publishing Co., Reading, MA, 1985.

M. M. Mano e C. Kime. Logic and Computer Design Fundamentals. Prentice-Hall International, Englewood Cliffs, N.J., 1997.

J. A. Marques e P. Guedes. Fundamentos de Sistemas Operativos. Editorial Presença, Lisboa, Portugal, 3 edição, 1994.

J. A. Marques e P. Guedes. Tecnologia dos Sistemas Distribu ídos. FCA–Editora de Informática, Lisboa, Portugal, 1998.

J.J. McConnell. Analysis of Algorithms: An Active Learning Approach. Jones and Bartlett Publishers, Sudbury, MA, 2 edição, 2008.

B. Meyer. Object Oriented Software Construction. IEEE Computer Society Press, NEW YORK, N.Y., 2 edição, 1997.

M. L. Modesto. Cozinha Tradicional Portuguesa. Editorial Verbo, Lisboa, Portugal, 1982.

J. W. Moore. Software Engineering Standards: A User's Road Map. IEEE

Computer Society, Los Alamitos, CA, 1998.

B. Moret. The Theory of Computation. Addison-Wesley Publishing Co., Reading, MA, 1997.

S. L. Pfleeger e J. M. Atlee. Software Engineering. Prentice-Hall Inc., Englewood Cliffs, N.J., 4 edição, 2010.

R. Ramakrishnan e J. Gehrke. Database Management Systems. McGraw-Hill Book Company, New York, N.Y., 2 edição, 2000.

B. Raphael. The Thinking Computer: Mind Inside Matter. W.H.

Freeman, S.

Francisco, CA, 1976.

J. A. Robinson. A machine-oriented logic based on the resolution principle.

Journal of the ACM, 12(1):23–41, 1965.

418

BIBLIOGRAFIA

S. Russell e P. Norvig. Artificial Intelligence: A Modern Approach. Prentice Hall, Englewood Cliffs, N.J., 3 edição, 2010.

S. Schach. Object-Oriented and Classical Software Engineering. McGraw-Hill Book Company, New York, N.Y., 6 edição, 2005.

G. M. Schneider, S. Weingart, e D. Perlman. An Introduction to Programming and Problem Solving with Pascal. John Wiley & Sons, New York, N.Y., 1978.

M. Schönfinkel. On the building blocks of mathematical logic. In Heijenoort, editor, From Frege to Gödel, A Source Book in Mathematical Logic, 1879–

1931. Harvard University Press, Cambridge, MA, 1977.

M. L. Scott. Programming Language Pragmatics. Morgan Kaufmann Publishers, San Francisco, CA, 2000.

C. Sernadas. Introdução à Teoria da Computação. Editorial Presença, Lisboa, Portugal, 1993.

D. Shell. A highspeed sorting procedure. Comm. of the ACM, 2(7):30–32, 1959.

A. Silberschatz, P. B. Galvin, e G. Gagne. Operating System Concepts. John Wiley & Sons, New York, N.Y., 6 edição, 2001.

S. Sing. The Code Book: The Secret History of Codes and Code-Breaking. Fourth Estate, London, UK, 1999.

M. Sipser. Introduction to the Theory of Computation. PWS Publishing Co., Boston, MA, 3 edição, 2012.

I. Sommerville. Software Engineering. Addison-Wesley Publishing Co., Reading, MA, 5 edic~ao, 1996.

H. S. Stone. Introduction to Computer Architecture. Science Research Associates, Inc., Chicago, IL, 1975.

A. S. Tanenbaum. Modern Operating Systems. Prentice-Hall Inc., Englewood Cliffs, N.J., 2 edic~ao, 2001.

A. L. Taylor. The wizard inside the machine. TIME, 123(16):42–49, 1984.

F. Turbak, D. Gifford, e M. A. Sheldon. Design Concepts in Programming Languages. MIT Press, Cambridge, MA, 2008.

BIBLIOGRAFIA

419

A.J. Turner. A summary of the acm/ieee-cs joint curriculum task force report: Computing curricula 1991. Communications of the ACM, 34(6):69–84, 1991.

L. B. Wilson e R. G. Clark. Comparative Programming Languages. Addison-Wesley Publishing Co., Reading, MA, 1988.

N. Wirth. Systematic Programming: An Introduction. Prentice-Hall Inc., Englewood Cliffs, N.J., 1973.

N. Wirth. Algorithms + Data Structures = Programs. Prentice-Hall Inc., Englewood Cliffs, N.J., 1976.

Índice

!=, 51

ADT, 262

*, 18, 39, 41, 42, 115, 126, 136

al-Khowarizm, Abu Ja'far Mohammed

+, 18, 39, 41, 42, 115, 126, 136

ibu-Musa, 6

-, 39, 41, 42

Alexandria, Heron de, 86

/, 41, 42

Algol, 96

//, 39

algoritmo, 6, 390, 405

:: =, 18

de Euclides, 85, 173

<, 51

alisa, 117

< =, 51

ambiente, 46

=, 44

global, 46, 47, 98

= =, 51

local, 76, 78

>, 51

análise

> =, 51

do problema, 306, 399

#, 66, 314

dos requisitos, 307

%, 39

and, 43

, 18

anonimato da representaç~ao, 272

Argand, Jean-Robert, 257

abordagem do topo para a base, 87, Arquitectura de Computadores, 396, 410

396

Arquitectura de Programas, 399

abstracç~ao, 78, 256, 308, 390, 396

árvore, 371

construç~ao de, 397

binária, 372

de controle, 397

binária de procura, 384

de dados, 256, 398

filha, 371

de máquinas, 397

folha, 371

procedimental, 78–80, 103, 256, 397

mãe, 371

actualiza ciclo, 368

ordenaç~ao por, 384

açúcar sintáctico, 52

raiz, 371

acumula, 182

ramo, 371

acumulador, 182

vazia, 371

420

ÍNDICE

421

ASCII, 127

controle do, 63

assinatura de um tipo, 264

corpo do, 63

atribuição, 44

infinito, 63

avalia, 332

lê-avalia-escreve, 32

passagem pelo, 63

Bachmann, Paul, 161

cifra

Backus, John, 27

cifra de substituição, 129

base de conhecimento, 403

class, 278

base de dados, 408

classe, 278, 285

de objectos, 409

hierarquia de, 291

Basic, 401

cliente (classe), 345

Becker, Joe, 127

Clos, 403

biblioteca, 104

coerção, 42

boa aprox, 189

Collins, Lee, 127

bom palpite, 88

comando, 33

Bombelli, Rafael, 257

comentário, 66, 314

Boole, George, 43

compilador, 395

break, 64, 120

compl iguais, 269

Brooks Jr., Frederick P., 318

compl zero, 268

C, viii, 12, 401

complexidade do algoritmo, 392

C++ , viii, 403

complexol, 267

cadeia de caracteres, 33, 35, 123

comportamento

vazia, 123

FIFO, 337

cadeia de caracteres de documentação,

LIFO, 321

123

computabilidade, 392

computador, 4

caixa (classe), 347

características, 4

calc termo, 93

condição, 51

calcula raiz, 87, 192

cálculo lambda, 179

conhecimento

carácter

declarativo, 91

de escape, 54

procedimental, 91

de pronto, 32, 60, 75

conjunto

Chomsky, Noam, 27

de chegada, 73

Church, Alonzo, 4, 179

de partida, 73

ciclo, 63, 190

constante, 33

contado, 119

construtores, 263

conta gen, 288, 290

422

ÍNDICE

conta jovem, 291

externa, 314

conta jovem com pin, 293

interna, 314

conta ordenado, 290

técnica, 313, 314

contração, 200

de utilização, 313

controle da complexidade, 308

domínio

cria compl, 258, 260

de um nome, 103

cria complexo, 267

estático, 103

crivo, 143

Crivo de Eratóstenes, 143

efeito, 56

Eiffel, 403

dados

encapsulação da informação, 271

persistentes, 221

engenharia da programação, 304

Davis, Mark, 127

engenharia informática, 2

de Pisa, Leonardo, 207

entidade

debugging, 311

com estado, 284

definição, 33, 74, 273

computacional, 33

recursiva, 20, 167, 191

recursiva, 167

del, 136

Eratóstenes, 143

depuração, 26, 310

erro

da base para o topo, 312

absoluto, 88

depurar, 310

relativo, 88

derivada, 187

semântico, 25, 312

derivada ponto, 186

sintático, 25, 311

desenvolvimento

escreve compl, 260

da solução, 399

estrutura

do topo para a base, 308

de blocos, 95

dicionário, 235

estruturas lineares, 321

chave, 235

Euclides, 84

valor, 235

evolução local, 197

vazio, 236

existem caixas com fila, 352

dict, 238

expansão, 200

dificuldade NP , 392

expressão, 33, 44

dificuldade P , 392

composta, 33, 35

Dijkstra, Edsger, 246, 316, 398

designatória, 72, 235

divide compl, 259

documentaç~ao

factorial, 83, 198, 201

de concepç~ao, 308

False, 35, 43

ÍNDICE

423

fib, 208, 210

pública, 100

Fibonacci, Leonardo, 207

recursiva, 171

ficheiro, 221, 410

valor devolvido por, 77

abertura de, 222

funç~ao em informática

fecho de, 225

vs. funç~ao matemática, 84

modo de escrita, 222

modo de leitura, 222

gest~ao

FIFO, 337

de comunicaç~oes, 411

da concorrência, 408

fila (classe), 341

fila, 337

de processos, 410

de prioridades, 369

de memória, 408, 411

Gödel, Kurt, 4

filtra, 181

filtro, 181

grafo, 246

arco, 246

float, 42

dirigido, 246

for, 119

rotulado, 246

Fortran, 401

FP, 402

caminho, 246

função, 72, 235

nó, 246

anónima, 179

gramática, 17

aplicação, 73

ambígua, 21

argumento, 72

guião, 57

número arbitrário de, 381

hanoi, 215

chamada, 73, 76

Haskell, 402

como valor de função, 186

help, 123

contradomínio, 72

herança, 292

corpo da, 74

heurística, 405

de Ackermann, 219

de avaliação, 55

if, 59

definição, 73

imag puro, 268

por abstração, 72

in, 115, 126, 136, 238

por compreensão, 72

indicador

por enumeração, 72

de escrita, 228

por extensão, 72, 235

de leitura, 224

derivada, 187

índice, 112, 113

domínio, 72

informação, 2

factorial, 83

informática, 3

indefinida, 73

instância, 278

424

ÍNDICE

instrução, 33

semântica, 24

break, 64, 120

sintaxe, 16

de atribuição, 44, 190

Lisp, 402

múltipla, 48

list, 136

simples, 45

lista, 135

de importação, 104

associativa, 235

for, 119

funcionais sobre, 181

if, 59

vazia, 136

raise, 89

listas paralelas, 155

try-except, 326

vazia, 65

manutenção de um programa, 318

máquina de inferência, 403

while, 63

máquina virtual, 394

int (como função), 42, 341

Inteligência Artificial, 405

Matemática Discreta, 392

interface gráfica, 407

matriz, 164

interpretador, 13, 394

mdc, 85

introduz notas, 245

met newton, 189

isinstance, 117, 267, 277

met intervalo, 184

metalinguagem, 24

Java, viii, 12, 403

método

init , 273

Kleene, Stephen C., 4

aproximado, 86, 91

Kramp, Christian, 83

de Curry, 194

lambda, 179

definição de, 273

Landau, Edmund, 161

do intervalo, 182

Ledgard, Henri, 309

de Newton, 188

lei de Murphy, 306

passagem de parametros, 139

Miranda, 402

len, 115, 126, 136, 238

LIFO, 321

ML, 402

linguagem

modelo, 2

assembly, 13

da cascata, 305

de alto nível, 13, 57, 394

do programa armazenado, 400

de programação, 4, 6, 12, 393

de von Neumann, 400

estruturada em blocos, 96

modificadores, 329

imperativa, 44

modularidade da solução, 308

máquina, 12, 57

módulo, 104

processador de, 13

mostra janela inicial, 368

ÍNDICE

425

mova, 213, 215

primo, 108, 143

mova disco, 215

pseudo-aleatório, 346

multimédia, 407

real, 34

multiplica compl, 259

triangular, 109

multiutilização, 410

objecto, 272

Naur, Peter, 27

definição de, 273, 289, 290

nome, 33, 44

definição de método, 273

composto, 45, 105, 224, 240, 274,

método, 273

276

obtem info, 245

domínio de, 103

open, 222

global, 102

operação

indexado, 45, 113

embutida, 35

livre, 102

escrita de dados, 55

local, 101

leitura de dados, 52

não local, 102

polimórfica, 297

reservado, 45

sobrecarregada, 41

simples, 45

operações

None, 378

básicas de um tipo, 263

not, 43

operador, 35

not in, 238

de atribuição, 44

notação

prioridades, 36

BNF, 17

operando, 35

científica, 34

or, 43

de Bachmann-Landau, 161

ordem de crescimento, 157

O, 392

ordena árvore, 387

o, 392

ordenação

Omaiúsculo, 161

por árvore, 384

Ω , 392

por borbulhamento, 152

pós-fixa, 331

por selecção, 154

Θ , 392

quick sort, 165

novo palpite, 88

Shell, 153

numero, 267

número, 33

p imag, 259, 260, 267

aleatório, 346

p real, 259, 260, 267

de Fibonacci, 208

paradigma de programação, 190, 400

inteiro, 34

paragrafação, 59

426

ÍNDICE

parâmetro

linear, 201

concreto, 75

procura

formal, 73

binária, 148

número arbitrário de, 381

linear, 147

Pascal, 401

sequencial, 147

passagem

programa, 2–4, 11, 57, 305, 390

por referência, 142

fonte, 395, 398

por valor, 140

objecto, 395

piatorio, 194

programação, 3, 390

pilha (classe), 329

com objectos, 402

pilha, 321

em lógica, 403

pixel, 356

funcional, 190, 401

polimorfismo, 297

imperativa, 190, 400

ponto (como classe), 364

paradigma de, 190, 400

Post, Emil Leon, 4

Prolog, 404

potencia, 83, 94, 97, 191, 198, 203

pseudónimo, 138

potência rápida, 217

potencia aux, 95

quadrado, 79, 186

potencia estranha, 102

raciocínio, 406

potencia rapida, 217

raiz, 89, 184, 191

potencia tambem estranha, 102

raise, 89

predicado, 51

range, 120

embebido, 51

reconhecedores, 263

print, 56

recurs~ao

prioridade

definiç~ao, 167

de operador, 36

em árvore, 207, 209

problema

em funç~oes, 206

grau de dificuldade de, 162

em processos, 207

processa resultados, 351, 368

parte

processador de linguagem, 393, 395

básica, 173

processamento interactivo, 32

recursiva, 173

processo

partes da definição, 173

computacional, 3, 197, 390

refinação por passos, 308

iterativo, 206, 207

regime de multitarefa, 410

linear, 206

representação

recursivo, 200, 207

declarativa, 404

em árvore, 207

ÍNDICE

427

em vírgula flutuante, 39

string, 35, 123

externa, 33, 41, 112, 260, 324, 339 subclasse, 288–290

interna, 33, 260

herança, 289, 290

procedimental, 404

subtraí compl, 259

return, 76

suf pequeno, 93

round, 42

suf perto, 184

seletores, 263

tabela, 135

semântica, 24, 393

tabela de dispersão, 235

de uma linguagem, 24

Teoria da Computação, 392, 393

sentinela, 65

teste

sequênciação, 58

de aceitação, 317

sessão em Python, 32

casos de, 316

Shell, Donald, 153

de integração, 317

símbolo

de módulo, 310

continuação em Python, 93

de sistema, 317

não terminal, 17

unitário, 310

terminal, 18

testes, 264

simula supermercado, 349, 365

tipo

simulação, 343

de informação, 37

sin, 92

domínio do, 37

sintaxe, 16, 393

elementar, 37

de uma linguagem, 16

elemento do, 37

sistema

em Python

computacional, 305

bool, 43

de ficheiros, 411

file, 221

de gestão da base de dados, 408

float, 39

de informação, 408

int, 39

operativo, 409

list, 135

Smalltalk, 403

None, 378

soma compl, 259

str, 35, 123

soma elementos, 117, 120, 121, 124,

tuple, 111

200, 205

estruturado, 38

somatorio, 179, 180

imutável, 114, 124

str (como função), 299

mutável, 135, 235

str (como função), 341

tipos abstractos de informação, 262

Stratchey, Christopher, 174

construtores, 263, 264

428

ÍNDICE

modificadores, 329

reconhecedores, 263, 264

selectores, 263, 264

testes, 264, 265

transformador, 341

Torre de Hanói, 212

transf newton, 189

transforma, 181

transformada de Newton, 188

transformador, 181, 341

- de entrada, 265
- de saída, 265
- trata clientes, 349, 351, 367
- True, 35, 43
- tuple, 115
- tuplo, 111
- vazio, 112
- Turing, Alan, 4
- type, 277, 296
- Unicode, 127, 227
- valor lógico, 33, 35
- valor sentinela, 65
- van Rossum, Guido, viii
- variável, 44
- de estado, 206, 284
- inicialização, 65
- muda, 72
- vector, 135
- while, 63
- Zelle, John, 354
- Errata

Na página 13, linha 1, onde está “por 0 e por 1” deve estar “por 1 e por 0” (detectado por Tiago Mendes de Almeida Rechau)

Na página 14, linha 3, onde está “da precisão” deve estar “do rigor” (detectado por Bernardo Pascoal Figueiredo)

Na página 19, linha -3 onde está “o joão” deve estar “o João” (detectado por Inês Santos) Na página 25, última linha, onde está “Os erros sintácticos podem-se manifestar” deve estar

“Os erros semânticos podem-se manifestar” (detectado por Diogo Panaca Mendes) Na página 26, linha -4, onde está “de uma fora sistemática” deve estar “de uma forma sistemática” (detectado por Luís Jordão Alves) Na página 28, exercício 3, onde está BNF’ deve estar BNF (detectado por João Colaço) Na página 32, linha 3, onde está “mostrando no écran do resultado” deve estar “mostrando no écran o resultado” (detectado por Luís Jordão Alves) Na página 47, linha4, onde está “os nomes de todos” deve estar “os nomes de todas”

(detectado por Nuno Miguens Diogo)

Na página 47, nota de rodapé, onde está “name space” deve estar “namespace” (detectado por Miguel Pasadinhas)

Na página 54, linha 14, onde está “um expressão” deve estar “uma expressão” (detectado por João Romeiras Amado)

Na página 57, linha 22, onde está := deve estar :: = .

Na página 58, linhas 19, 21, 23 e 24, onde está := deve estar :: = .

Na página 61, linha 18, onde está “o o Python” deve estar “o Python” (detectado por Francisco Castro Henriques)

Na página 63, linha 19, onde está “estamos em presença” deve estar “estamos na presença”

(detectado por Yuliya Plotka)

Na página 66, linha 10, onde está “expresões” deve estar “expressões”.

Na página 67, linha 14, onde está “as temperatura” deve estar “as temperaturas” (detectado por Daniel Sil)

Na página 67, linha 17, onde está “um do dígitos” deve estar “um dos dígitos” (detectado por Daniel Sil)

Na página 71, linha 20, onde está “Pyton” deve estar “Python” (detectado por Márcio Santos) Na página 72, linha 16, onde está “expressões, símbolo” deve estar “expressões, o símbolo”

(detectado por Márcio Santos)

Na página 75, linha 1, onde está “termina a” deve estar “e termina a” (detectado por Márcio Santos)

Na página 75, penúltima linha, onde está “pelo quando” deve estar “pelo que quando”

(detectado por Márcio Santos)

Na página 77, linha 13, onde está “o que eles fazem” deve estar “o que elas fazem”

(detectado por Márcio Santos)

Na página 78, linha 11, onde está “escreveu” deve estar “escrever” (detectado por Anisa Shahidian)

Na página 80, linha 16, onde está “while n > 0:” deve estar “while n != 0:”

Na página 86, linha -10, onde está "função" deve estar “função”
(detectado por Bernardo Pascoal Figueiredo)

Na página 87, linha -2, onde está “o nossa” deve estar “a nossa”
(detectado por Francisco Castro Henriques)

Na página 88, linha 6, onde está “Teferência” deve estar “Refêrencia”
(detectado por Pedro Pinela)

Na página 89, linha 9, onde está “internos igual” deve estar “internos igual a” (detectado por Márcio Santos)

Na página 91, linha -9 onde está “modo e criar” deve estar “modo de criar” (detectado por João Filipe Coito Ribeiro)

Na página 91, linha -4, onde está “while n > 0:” deve estar “while n != 0:”

Na página 92, linha 4, onde está “n > 0” deve estar “n != 0”

Na página 92, linha 11, onde está “while e > 0:” deve estar “while e != 0:”

Na página 92, linha 16, onde está “while e < 0:” deve estar “while e != 0:”

= 0:”

Na página 93, linhas 16-17, onde está “quais condições” deve estar “quais as condições”

(detectado por Márcio Santos)

Na página 94, linha 2, onde está “while e > 0:” deve estar “while e != 0:”

Na página 94, linha 3, onde está “res = res * x” deve estar “res = res * b”

Na página 95, na linha 4 da Figura 3.5, onde está “while e > 0:” deve estar “while e

!= 0:”

Na página 95, na linha 5 da Figura 3.5, onde está “res = res * x” deve estar “res =

res * b”

Na página 96, linha 2, onde está “potencia-aux” deve estar “potencia_aux” (detectado por Márcio Santos)

Na página 104, linha 16, onde está “modul” deve estar “módulo” (detectado por Francisco Castro Henriques)

Na página 104, linha 19, onde está “funções m” deve estar “funções em” (detectado por Márcio Santos)

Na página 112, linha 9, onde está “tentamos a utilizar” deve estar “tentamos utilizar”

(detectado por Márcio Santos)

Na página 114, linha 14, onde está “levados s pensar” deve estar “levados a pensar”

(detectado por Márcio Santos)

Na página 114, linha -7, onde está “função recebe” deve estar “função que recebe”

(detectado por Márcio Santos)

Na página 115, linha 14 onde está “(1, 2, ((3,), ((4,),), 5), (6, ((7,),)))” deve estar “alisa((1, 2, ((3,), ((4,),), 5), (6, ((7,),)))”

(detectado por Márcio Santos)

Na página 116, linha 2, onde está “com argumento” deve estar “como argumento” (detectado por Márcio Santos)

Na página 116, segunda linha do primeiro parágrafo, onde está “um elementos” deve estar

“um elemento” (detectado por Anisa Shahidian) Na página 116, linha -1, onde está “ao valor” deve estar “o valor” (detectado por Márcio Santos)

Na página 117, linha 10, onde está “instrução composts” deve estar “instrução composta”

(detectado por Adriana Domingos)

Na página 120, linha 7, onde está “tem” deve estar “têm” (detectado por Márcio Santos) Na página 120, linha 17, onde está “processar um tuplo que é alterado durante o processamento” deve estar “processar uma variável, cujo tuplo associado é alterado durante o processamento” (detectado por Miguel Neves Pasadinhas) Na página 120, linha -7, onde está “delimitados por plicas” deve estar “delimitada por plicas”

(detectado por João Carrapiço)

Na página 121, linha 17, onde está “dessa essa cadeia” deve estar “dessa cadeia”

(detectado por Pedro Pinela)

Na página 130, linha 4, onde está “uma tuplo” deve estar “um tuplo” (detectado por Margarida Correia)

Na página 130, linha -3, onde está “junção” deve estar “função”

Na página 136, linha 11, onde está “cria ao nome” deve estar “cria o nome” (detectado por Márcio Santos)

Na página 139, linha 2, onde está “como se mosta na Figura 5.7” deve estar “como se mostra na Figura 5.7” (detectado por Márcio Santos) Na página 139, linha 5, onde está “o que atribui estas valores” deve estar “o que atribui estes valores”

Na página 141, linha -7, onde está “seleccionado for menor que” deve estar “seleccionado não for maior que” (detectado por João Gonçalves Ceia) Na página 142, linha 3, onde está “inferiores a” deve estar

“inferiores ou iguais a” (detectado por João Gonçalves Ceia)

Na página 142, linha 16, onde está “while lista[i] < sqrt(n):” deve estar “while lista[i] <= sqrt(n):” (detectado por João Gonçalves Ceia)
Na página 142, linha -4, onde está “inferiores a” deve estar “não superiores a” (detectado por João Gonçalves Ceia)

Na página 142, linha -3, onde está “no ciclo while” deve estar “no ciclo while” (detectado por Alexandre Ferreira)

Na página 143, linha 13, onde está “função, pos que”, deve estar “função, pos, que”

Na página 143, linha 14, onde está “neste elemento” deve estar “deste elemento” (detectado por Carlos Carvalho)

Na página 143, linha -6, onde está “do o elemento” deve estar “do que o elemento”

(detectado por Vicente Rocha)

Na página 144, linha 14, onde está “while lista[i] < sqrt(n):” deve estar “while lista[i] <= sqrt(n):” (detectado por João Gonçalves Ceia)
Na página 158, linha 4, onde está “De modo obter” deve estar “De modo a obter” (detectado por Márcio Santos)

Na página 169, linha -11, onde está “claramente uma definição recursiva” deve estar

“claramente a uma definição recursiva” (detectado por Márcio Santos)
Na página 176, linha -8, onde está “a soma dos factorais” deve estar “a soma dos factoriais”

(detectado por Márcio Santos)

Na página 181, linha -11, onde está “funções” deve estar “funções”
(detectado por Márcio Santos)

Na página 187, linha 3, onde está “directamente a esta definição”
deve estar “directamente esta definição”(detectado por Márcio Santos)
Na página 196, última linha, onde “expresões” deve estar “expressões”
(detectado por Márcio Santos)

Na página 198, linha -11, onde está “todas eles geram” deve estar
“todas elas geram”

(detectado por Márcio Santos)

Na página 199, penúltima linha, onde está “pox” deve estar “prox”
(detectado por Márcio Santos)

Na página 203, linhas -12 a -7, onde está “-“, deve estar “_”
(detectado por Márcio Santos) Na página 205, linha -15, onde está
“tanto função potencia” deve estar “tanto função potencia” (detectado
por Márcio Santos)

Na página 211, linha 3 onde está “Mvimentar” deve estar
“Movimentar” (detectado por Márcio Santos)

Na página 214, primeira linha da Figura 7.5, onde está “torre-de-
hanoi()” deve estar

“hanoi()”

Na página 214, legenda da Figura 7.5, onde está “torre-de-hanoi()” deve estar

“hanoi()” (detectado por Márcio Santos)

Na página 214, linha 2, onde está “discos discos” deve estar “discos” (detectado por Márcio Santos)

Na página 222, linha 20, onde está “designamemos” deve estar “designaremos” (detectado por João Filipe Coito Ribeiro)

Na página 222, linha 20, onde está “que que” deve estar “que” (detectado por Hugo Ferreira) Na página 226, linha 12, onde está “que que” deve estar “que” (detectado por Daniel Tanque)

Na página 227, linha -13, onde está “aberto para leitura” deve estar “aberto para escrita”

(detectado por Pedro Pinela)

Na página 228, a linha 19, “print('abccde\nfgh 5 25\nijk\n\n\nlmn')” deve ser removida (detectado por Miguel Neves Pasadinhas) Na página 230, linha -10, onde está “que que” deve estar “que”

Na página 230, linha -7, onde está “que que” deve estar “que”

Na página 240, linha 2 das nota de rodapé, onde está “que que” deve estar “que” (detectado por Francisco Maria Calisto)

Na página 241, linha 20, onde está “conta_palavras” deve estar “conta_letras”

(detectado por Mariana Gaspar Fernandes)

Na página 245, linha -9, onde está “pos = 0;” deve estar “pos = 0”
(detectado por Pedro Pinela)

Na página 248, figura 9.3 (b) a distância calculada associada ao nó B não deve ser (15) mas sim (10) (detectado por Pedro Pinela)

Na página 249, figura 9.4 (a) a distância calculada associada ao nó B não deve ser (15) mas sim (10) (detectado por Pedro Pinela)

Na página 249, figura 9.4 (b) a distância calculada associada ao nó B não deve ser (15) mas sim (10) (detectado por Pedro Pinela)

Na página 257, linha 17, onde está “números números” deve estar “números” (detectado por Francisco Maria Calisto)

Na página 258, linha -2, onde está “aborgagem” deve estar “abordagem” (detectado por Vasco Almeida)

Na página 266, última linha, onde está “valoir” deve estar “valor”
(detectado por João Carrapiço)

Na página 269, linha 6, onde está “iguais” deve estar “igual”
(detectado por João Martins) Na página 271, linha -3, onde está
“corresponde tipo” deve estar “corresponde ao tipo”

(detectado por Márcio Santos)

Na página 273, linha 14, onde está “correspondentes e esse” deve estar “correspondentes a esse” (detectado por Márcio Santos)

Na página 278, linha 9, onde está “Python se usa” deve estar “Python usa” (detectado por Pedro Pinela)

Na página 279, legenda da Figura 10.4, onde está “comp” deve estar “compl” (detectado por Márcio Santos)

Na página 286, Figura 11.1, onde está “conta-01” deve estar “conta_01” (detectado por Márcio Santos)

Na página 286, Figura 11.1, onde está “conta-02” deve estar “conta_02” (detectado por Márcio Santos)

Na página 286, linha 4, onde está “conta-01” deve estar “conta_01” (detectado por Márcio Santos)

Na página 286, linha 4, onde está “conta-02” deve estar “conta_02” (detectado por Márcio Santos)

Na página 292, Figura 11.2, onde está “conta-ord-01” deve estar “conta_ord_01”

(detectado por Márcio Santos)

Na página 292, Figura 11.2, onde está “conta-jov-01” deve estar “conta_jov_01”

(detectado por Márcio Santos)

Na página 302, linha 9, onde está “asída” deve estar “saída”
(detectado por Maria Teresa Coelho)

Document Outline

- [Capítulo 1](#)
- [Capítulo 2](#)
- [Capítulo 3](#)
- [Capítulo 4](#)
- [Capítulo 5](#)
- [Capítulo 10](#)
- [Capítulo 11](#)